

General Computer Science
320201 GenCS I & II Lecture Notes

Michael Kohlhase

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

May 22, 2015

Preface

This Document

This document contains the course notes for the course General Computer Science I & II held at Jacobs University Bremen¹ in the academic years 2003-2015.

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

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

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

Knowledge Representation Experiment:

This document is also an experiment in knowledge representation. Under the hood, it uses the \LaTeX package [Koh08, Koh15], a $\text{\TeX}/\text{\LaTeX}$ 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.

¹International 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 notion 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 \LaTeX . 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 Ioniță, Semir Elezović, Dimitar Asenov, Alen Stojanov, Felix Schlesinger, Ștefan 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, Rubin Deliallisi, Mariia Gladkova.

Recorded Syllabus for 2014/15

In this document, we record the progress of the course in the academic year 2014/2015 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 1.	through with admin	9	8
2	Sep 2.	at end of motivation	29	19
3	Sep 8.	proofs with Peano axioms	35	27
4	Sep 9.	operators on unary natural numbers	41	31
5	Sep 15.	operations on sets	??	??
6	Sep 16.	Functions	54	39
7	Sep 22.	operations on functions	59	42
8	Sep 23.	SML function by cases	68	48
8	Sep 29.	higher-order functions	70	49
9	Sep 30.	SML data types	86	58
10	Oct 6.	constructor terms & substitutions	100	69
11	Oct 7.	recursion/computation on ADTs	111	75
12	Oct 13.	I/O and exceptions	??	??
13	Oct 14.	Codes, up to string codes	132	91
	Oct 27	Midterm		
14	Oct 28	UTF encodings	139	95
15	Nov 3.	Truth Tables	148	102
16	Nov 4.	Landau Sets	155	106
17	Nov 10.	Cost Bounds for Boolean Expressions	160	109
18	Nov 11.	Quine McCluskey	179	117
19	Nov 17.	Intro to Propositional Logic	188	124
20	Nov 18.	The miracle of logics	192	128
21	Nov 24.	Properties of the Hilbert-Calculus	199	132
22	Nov 25.	Natural Deduction for Mathtalk	210	138
23	Dec 1.	Tableaux & Resolution	227	150
24	Dec 2.	Intellectual Property, Copyright, & Information Privacy	242	162

Recorded Syllabus Spring Semester 2015:

#	date	until	slide	page
1	Feb 2.	Labeled Graphs	253	177
2	Feb 4.	Balanced Binary Trees & sizes	264	183
3	Feb 9.	Arithmetics for PNS	274	191
4	Feb 10.	Adders	285	197
5	Feb 16.	Add/Subtraction Unit	298	205
6	Feb 17.	CPU Architecture	313	217
7	Feb 23.	Assembler Language	318	220
8	Feb 24.	Virtual machine & Implementation	332	228
9	Mar 2.	Compiling SW statements	341	233
10	Mar 3.	VM with Procedures (Simulation)	348	237
11	Mar 9.	Compiling μ ML	362	251
12	Mar 10.	Universal Turing Machine/Halting Problem	373	258
13	Mar 16.	Internet Protocol Suite	378	264
14	Mar 17.	SMTP over telnet	389	271
	Mar 23.	midterm		
15	Mar 24.	WWW Overview	396	274
	Mar 30/31.	spring break		
16	Apr 7.	CSS	404	279
17	Apr 13.	WebApps & State	409	282
18	Apr 14.	Web Search	419	289
19	Apr 20.	public key encryption	438	299
20	Apr 21.	XPath	456	308
21	Apr 27.	Breadth-first search	??	??
22	Apr 28.	Semantic Web	465	313
23	May 4.	A* search	541	346
24	May 5.	local search/genetic algorithms	564	354
23	May 11.	PROLOG ideas	572	360
24	May 12.	PROLOG & Look Back		

Here the syllabus of of the last academic year for reference, the current year should be similar; see the course notes of last year available for reference at <http://kwarc.info/teaching/GenCS2/notes2013-14.pdf>.

#	date	until
1	Sep 2.	through with admin
2	Sep 3.	at end of motivation
3	Sep 9.	recap of the first week
4	Sep 10.	introduced proofs with Peano axioms
5	Sep 16.	covered mathtalk & Alphabets
6	Sep 17.	Relations
7	Sep 23.	properties of functions
8	Sep 24.	SML component selection
8	Sep 30.	higher-order functions
9	Oct 1.	datatypes
10	Oct 7.	computation on ADTs
11	Oct 8.	constructor terms & substitutions
	Oct 14	Midterm
12	Oct 15.	final Abstract Interpreter
13	Oct 28.	recursion relation & Fibonacci
14	Oct 29.	I/O and exceptions
15	Nov 4.	Codes, up to Morse Code
16	Nov 5.	UTF encodings
17	Nov 11.	Boolean Expressions
	Nov 12.	Midterm 2
18	Nov 18.	Boolean Functions
19	Nov 19.	Elementary Complexity Theory
20	Nov 25.	Quine McCluskey
21	Nov 26.	Intro to Propositional Logic
22	Dec 2.	The miracle of logics
23	Dec 3.	Natural Deduction for Mathtalk

Recorded Syllabus Spring Semester 2014:

#	date	until
1	Feb 3.	Graph Paths
2	Feb 4.	Balanced Binary Trees
3	Feb 10.	Universality of NAND and NOR
4	Feb 11.	Carry Chain adder
5	Feb 17.	Two's complement numbers
6	Feb 18.	RAM
7	Feb 24.	Basic Assembler
8	Feb 25.	Virtual machine basics
9	Mar 3.	Implementing VM arithmetics
10	Mar 4.	Compiling SW into VM
11	Mar 10.	Realizing Call Frames
12	Mar 11.	Compiling μ ML
13	Mar 17.	Universal Turing Machine/Halting Problem
14	Mar 18.	Network Interface Controllers
15	Mar 24.	SMTP over telnet
16	Mar 25.	URIs, URLs, URNs
17	Mar 31.	CSS
18	Apr 1.	Dynamic HTML
19	Apr 7.	Web Search: Queries
19	Apr 22.	public key encryption
20	Apr 28.	XPath
21	Apr 29.	Breadth-first search
22	May 5.	A^* search
23	May 12.	PROLOG ideas
24	May 13.	PROLOG & Look Back

Contents

Preface	ii
This Document	ii
Course Concept	ii
Course Contents	ii
Acknowledgments	iii
Recorded Syllabus for 2014/15	v
1 Getting Started with “General Computer Science”	1
1.1 Overview over the Course	1
1.2 Administrativa	3
1.2.1 Grades, Credits, Retaking	3
1.2.2 Homeworks, Submission, and Cheating	5
1.2.3 Resources	8
2 Motivation and Introduction	11
2.1 What is Computer Science?	11
2.2 Computer Science by Example	12
2.3 Other Topics in Computer Science	18
2.4 Summary	19
I Representation and Computation	21
3 Elementary Discrete Math	23
3.1 Mathematical Foundations: Natural Numbers	23
3.2 Reasoning about Natural Numbers	26
3.3 Defining Operations on Natural Numbers	29
3.4 Talking (and writing) about Mathematics	32
3.5 Naive Set Theory	34
3.6 Relations and Functions	37
3.7 Standard ML: Functions as First-Class Objects	43
3.8 Inductively Defined Sets and Computation	52
3.9 Inductively Defined Sets in SML	56
3.10 Abstract Data Types and Ground Constructor Terms	62
3.11 A First Abstract Interpreter	64
3.12 Substitutions	68
3.13 Terms in Abstract Data Types	70
3.14 A Second Abstract Interpreter	71
3.15 Evaluation Order and Termination	73

4	More SML	77
4.1	Recursion in the Real World	77
4.2	Programming with Effects: Imperative Features in SML	79
4.2.1	Input and Output	80
4.2.2	Programming with Exceptions	81
5	Encoding Programs as Strings	87
5.1	Formal Languages	87
5.2	Elementary Codes	89
5.3	Character Codes in the Real World	92
5.4	Formal Languages and Meaning	96
6	Boolean Algebra	99
6.1	Boolean Expressions and their Meaning	99
6.2	Boolean Functions	103
6.3	Complexity Analysis for Boolean Expressions	105
6.3.1	The Mathematics of Complexity	106
6.3.2	Asymptotic Bounds for Costs of Boolean Expressions	108
6.4	The Quine-McCluskey Algorithm	111
6.5	A simpler Method for finding Minimal Polynomials	118
7	Propositional Logic	121
7.1	Boolean Expressions and Propositional Logic	121
7.2	A digression on Names and Logics	125
7.3	Calculi for Propositional Logic	126
7.4	Proof Theory for the Hilbert Calculus	129
7.5	A Calculus for Mathtalk	135
7.5.1	Propositional Natural Deduction Calculus	135
8	Machine-Oriented Calculi	141
8.1	Calculi for Automated Theorem Proving: Analytical Tableaux	141
8.1.1	Analytical Tableaux	141
8.1.2	Practical Enhancements for Tableaux	145
8.1.3	Soundness and Termination of Tableaux	147
8.2	Resolution for Propositional Logic	148
II	Interlude for the Semester Change	151
9	Legal Foundations of Information Technology	153
9.1	Intellectual Property, Copyright, and Licensing	153
9.1.1	Copyright	155
9.1.2	Licensing	158
9.2	Information Privacy	161
10	Welcome Back and Administrativa	165
10.1	Recap from General CS I	166
III	How to build Computers and the Internet (in principle)	169
11	Combinational Circuits	173
11.1	Graphs and Trees	173
11.2	Introduction to Combinatorial Circuits	180
11.3	Realizing Complex Gates Efficiently	182

11.3.1	Balanced Binary Trees	183
11.3.2	Realizing n -ary Gates	185
12	Arithmetic Circuits	189
12.1	Basic Arithmetics with Combinational Circuits	189
12.1.1	Positional Number Systems	189
12.1.2	Adders	192
12.2	Arithmetics for Two's Complement Numbers	199
12.3	Towards an Algorithmic-Logic Unit	205
13	Sequential Logic Circuits and Memory Elements	207
13.1	Sequential Logic Circuits	207
13.2	Random Access Memory	209
13.3	Units of Information	212
14	Computing Devices and Programming Languages	215
14.1	How to Build and Program a Computer (in Principle)	215
14.2	A Stack-based Virtual Machine	221
14.2.1	A Stack-based Programming Language	222
14.2.2	Building a Virtual Machine	224
14.3	A Simple Imperative Language	229
14.4	Basic Functional Programs	235
14.4.1	A Bare-Bones Language	235
14.4.2	A Virtual Machine with Procedures	236
14.4.3	Compiling Basic Functional Programs	247
14.5	Turing Machines: A theoretical View on Computation	251
15	The Information and Software Architecture of the Internet and World Wide Web	261
15.1	Overview	261
15.2	Internet Basics	264
15.3	Basic Concepts of the World Wide Web	272
15.3.1	Addressing on the World Wide Web	273
15.3.2	Running the World Wide Web	274
15.3.3	Multimedia Documents on the World Wide Web	277
15.4	Web Applications	279
15.5	Introduction to Web Search	284
15.6	Security by Encryption	289
15.6.1	Introduction to Crypto-Systems	289
15.6.2	Public Key Encryption	292
15.6.3	Internet Security by Encryption	300
15.7	An Overview over XML Technologies	306
15.8	The Semantic Web	309
IV	Search and Declarative Computation	315
16	Problem Solving and Search	319
16.1	Problem Solving	319
16.2	Search	324
16.3	Uninformed Search Strategies	327
16.4	Informed Search Strategies	342
16.4.1	Greedy Search	343
16.4.2	A-Star Search	346

16.4.3 Finding Good Heuristics	349
16.5 Local Search	350
17 Logic Programming	357
17.1 Introduction to Logic Programming and PROLOG	357
17.2 Programming as Search	361
17.3 Logic Programming as Resolution Theorem Proving	366
V A look back; What have we learned?	369

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

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]

Plot of “General Computer Science”

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


©: Michael Kohlhase
1


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 Jacobs¹ 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

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

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.

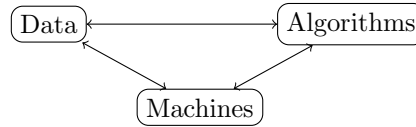


Figure 1.1: The three key ingredients of Computer Science

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 we 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.)



©: Michael Kohlhase

2





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)



©: Michael Kohlhase

3



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 instructor.

1.2.2 Homeworks, Submission, and Cheating

Homework assignments

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



©: Michael Kohlhase

4



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.

Homework Submissions, Grading, Tutorials

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



©: Michael Kohlhase

5



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 interests, and optimize the functioning of the society as a whole. This is also true for the community of scientists worldwide. This society is special, since it balances intense cooperation on joint issues with fierce competition. Most of the rules are largely unwritten; you are expected to follow them anyway. The code of academic integrity at Jacobs is an attempt to put some of the aspects into writing.

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

The Code of Academic Integrity

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


©: Michael Kohlhase
6


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 reputation². 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.

²Of 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



©: Michael Kohlhase

7



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



©: Michael Kohlhase

8



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.

Textbooks, Handouts and Information, Forum

- ▷ No required textbook, but course notes, posted slides
- ▷ Course notes in PDF will be posted at <http://old.kwarc.info/teaching/GenCS1>
- ▷ Everything will be posted on PantaRhei (Notes+assignments+course forum)
 - ▷ announcements, contact information, course schedule and calendar
 - ▷ discussion among your fellow students (careful, I will occasionally check for academic integrity!)
 - ▷ <http://panta.kwarc.info> (use your Jacobs login)
 - ▷ if there are problems send e-mail to course-gencs-tas@jacobs-university.de




©: Michael Kohlhase
9


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.

Software/Hardware tools

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


©: Michael Kohlhase
10


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 courses materials as a research object as well. This gives you an additional resource, but may affect the shape of the courses 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?**

©: Michael Kohlhase12

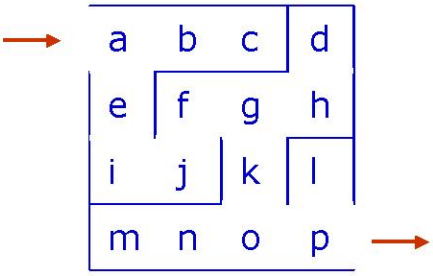
There are of course various ways to build such a a maze; one would be to ask the students from biology to come and plant some hedges, and have them re-plant them until the maze meets our criteria. A better way would be to make a plan first, i.e. to get a large piece of paper, and draw a maze before we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

An Answer:

Let's hack

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**.

SOME RIGHTS RESERVED ©: Michael Kohlhase 16 JACOBS UNIVERSITY

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 distinguish representations from the objects they represent, we will often be sloppy in our language, and rely on the ability of the reader to distinguish the levels.

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

Why math?

- ▷ **Q:** Why is it useful to formulate the problem so that mazes are room sets/pairs?
- ▷ **A:** Data structures are typically defined as mathematical structures.
- ▷ **A:** Mathematics can be used to reason about the correctness and efficiency of data structures and algorithms.
- ▷ **A:** Mathematical structures make it easier to **think** — to abstract away from unnecessary details and avoid “hacking”.



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

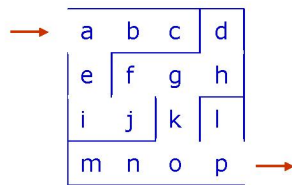
As mathematical models have very few properties (this is deliberate, so that we can understand all of them), we can use them as models for many concrete, real-world situations.

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

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

Mazes as Graphs

- ▷ **Definition 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

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



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.

SOME RIGHTS RESERVED ©: Michael Kohlhase 19 JACOBS UNIVERSITY

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**.

SOME RIGHTS RESERVED ©: Michael Kohlhase 20 JACOBS UNIVERSITY

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!

SOME RIGHTS RESERVED ©: Michael Kohlhase 21 JACOBS UNIVERSITY

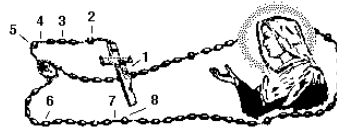
So, we know what we are looking for, we can think about a program that would find spanning trees given a set of nodes in a graph. But since we are still in the process of “thinking about the problems” we do not want to commit to a concrete program, but think about programs in the

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.



©: Michael Kohlhase

22



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

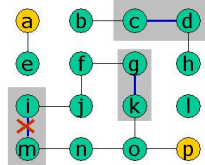
- ▷ When adding a wall to the tree, how do we detect that it won't create a cycle?
- ▷ When adding wall $\langle x, y \rangle$, we want to know if there is already a path from x to y in the tree.
- ▷ In fact, there is a fast algorithm for doing exactly this, called "Union-Find".

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 do some preliminary computations.

Performance and Scaling

- ▷ Suppose we have three algorithms to choose from. (which one to select)
- ▷ Systematic analysis reveals performance characteristics.
- ▷ **Example 2.2.13** For a problem of size n (i.e., detecting cycles out of n nodes) we have

	performance		
size	linear	quadratic	exponential
n	$100n \mu s$	$7n^2 \mu s$	$2^n \mu s$
1	100 μs	7 μs	2 μs
5	.5 ms	175 μs	32 μs
10	1 ms	.7 ms	1 ms
45	4.5 ms	14 ms	1.1 Y
100
1 000
10 000
1 000 000

©: Michael Kohlhase 25

What?! One year?

- ▷ $2^{10} = 1024$ ($\approx 1024 \mu s, 1 ms$)
- ▷ $2^{45} = 35\,184\,372\,088\,832$ ($3.5 \times 10^{13} \mu s = 3.5 \times 10^7 s \sim 1.1 Y$)
- ▷ **Example 2.2.14** we denote all times that are longer than the age of the universe with –

	performance		
size	linear	quadratic	exponential
n	$100n \mu s$	$7n^2 \mu s$	$2^n \mu s$
1	100 μs	7 μs	2 μs
5	.5 ms	175 μs	32 μs
10	1 ms	.7 ms	1 ms
45	4.5 ms	14 ms	1.1 Y
i 100	100 ms	7 s	$10^{16} Y$
1 000	1 s	12 min	–
10 000	10 s	20 h	–
1 000 000	1.6 min	2.5 mon	–

©: Michael Kohlhase 26

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.

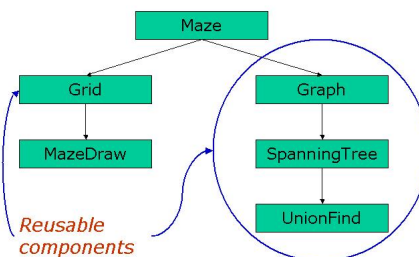
Is it correct?

- ▷ 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. However, 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.



©: Michael Kohlhase

29



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

Part I

Representation and Computation

Chapter 3

Elementary Discrete Math

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

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

Let’s start with the math!

Discrete Math for the moment

- ▷ Kenneth H. Rosen *Discrete Mathematics and Its Applications*, McGraw-Hill, 1990 [Ros90].
- ▷ Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, 1998 [LP98].
- ▷ Paul R. Halmos, *Naive Set Theory*, Springer Verlag, 1974 [Hal74].



©: Michael Kohlhase

30



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 a 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**.



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 explicit, 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, †1932**)
- ▷ **Axiom 3.1.6 (P1)** “ ” (aka. “zero”) is a unary natural number. “ ” (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.
 - ▷ “ ” is a unary natural number (axiom P1)
 - ▷ / is a unary natural number (axiom P2 and 1.)
 - ▷ // is a unary natural number (axiom P2 and 2.)
 - ▷ /// is a unary natural number (axiom P2 and 3.)
- ▷ **Definition 3.2.3** We call a sequence of **inference**s 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 concrete unary natural numbers.)
- ▷ Remember the two rules we had for dealing with unary natural numbers
- ▷ **Idea:** represent a number by the trace of the rules we applied to construct it. (e.g. $////$ is represented as $s(s(s(s(o))))$)
- ▷ **Definition 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)


©: Michael Kohlhase
36


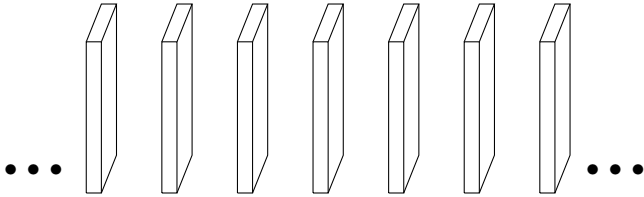
This systematic representation of natural numbers by expressions becomes very powerful, if we mix it with the practice of giving names to generic objects. These names are often called variables, when used in expressions.



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, \dots be a linear sequence of dominos, such that for any unary natural number i we know that
 - ▷ the distance between S_i and $S_{s(i)}$ is smaller than the height of S_i ,
 - ▷ S_i is much higher than wide, so it is unstable, and
 - ▷ 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.




©: Michael Kohlhase
37


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” \square

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) \square

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”. \square



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 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: $n \oplus 0 = n$

▷ 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 $s(s(o)) \oplus o$ (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 **P3** (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 intuitions.

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 $\exp(n, o) = s(o)$ and $\exp(n, s(m)) = n \odot \exp(n, m)$.

▷ **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.

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

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 idiom**s 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” 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 as a 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

- ▷ $o \in \mathbb{N}_1$ (zero is a unary natural number)
- ▷ $\forall n \in \mathbb{N}_1.s(n) \in \mathbb{N}_1 \wedge n \neq s(n)$ (\mathbb{N}_1 closed under successors, distinct)

$\triangleright \neg(\exists n \in \mathbb{N}_1. 0 = s(n)) \quad \text{(zero is not a successor)}$ $\triangleright \forall n \in \mathbb{N}_1. \forall m \in \mathbb{N}_1. n \neq m \Rightarrow s(n) \neq s(m) \quad \text{(different successors)}$ $\triangleright \forall P. (P(0) \wedge (\forall n \in \mathbb{N}_1. P(n) \Rightarrow P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}_1. P(m)) \quad \text{(induction)}$
 ©: Michael Kohlhase 43 

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, \dots, v_n as arguments, and the definiens is an expression in these variables. When the new operator is applied to arguments a_1, \dots, 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 :

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

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

Definition 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 as \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.

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

The Greek, Curly, and Fraktur Alphabets \rightsquigarrow Homework

▷ **Homework:** learn to read, recognize, and write the Greek letters

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

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

▷ BTW, we will also use the curly Roman and “Fraktur” alphabets:

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



To be able to *read and understand* math and computer science texts profitably it is 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)

 ©: Michael Kohlhase 45 

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



a
 $A \ni b$
 b

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

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

▷ **Relations between Sets**

- ▷ **set equality:** $(A \equiv B) :\Leftrightarrow (\forall x. x \in A \Leftrightarrow x \in B)$
- ▷ **subset:** $(A \subseteq B) :\Leftrightarrow (\forall x. x \in A \Rightarrow x \in B)$
- ▷ **proper subset:** $(A \subset B) :\Leftrightarrow (A \subseteq B) \wedge (A \neq B)$
- ▷ **superset:** $(A \supseteq B) :\Leftrightarrow (\forall x. x \in B \Rightarrow x \in A)$
- ▷ **proper superset:** $(A \supset B) :\Leftrightarrow (A \supseteq B) \wedge (A \neq B)$

 ©: Michael Kohlhase 46 

We want to have some operations on sets that let us construct new sets from existing ones. Again, we 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:** $A \setminus B := \{x \mid x \in A \wedge x \notin B\}$
- ▷ the **power set:** $\mathcal{P}(A) := \{S \mid S \subseteq A\}$
- ▷ the **empty set:** $\forall x, x \notin \emptyset$
- ▷ **Cartesian product:** $A \times B := \{\langle a, b \rangle \mid a \in A \wedge b \in B\}$, call $\langle a, b \rangle$ **pair**.
- ▷ **n -fold Cartesian product:** $A_1 \times \dots \times A_n := \{\langle a_1, \dots, a_n \rangle \mid \forall i, 1 \leq i \leq n \Rightarrow a_i \in A_i\}$, call $\langle a_1, \dots, a_n \rangle$ an **n -tuple**
- ▷ **n -dim Cartesian space:** $A^n := \{\langle a_1, \dots, a_n \rangle \mid 1 \leq i \leq n \Rightarrow a_i \in A\}$, call $\langle a_1, \dots, a_n \rangle$ 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)*
 - ▷ $\#(A \cup B) \leq \#(A) + \#(B)$ *(⚠ 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 identities 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 $\{O\}, \{\{O\}\}, \{O, \{O\}\}, \dots \in \mathcal{S}$,
 - ▷ contains all unions, intersections, power sets,
 - ▷ contains itself: $\mathcal{S} \in \mathcal{S}$ (scary!)
- ▷ Let’s make \mathcal{S} less scary


©: Michael Kohlhase
49


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)


©: Michael Kohlhase
50


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** $R^{-1} := \{\langle y, x \rangle \mid \langle x, y \rangle \in R\}$ is the **converse** relation of R .
- ▷ **Note:** $R^{-1} \subseteq B \times A$.
- ▷ The **composition** of $R \subseteq A \times B$ and $S \subseteq B \times C$ is defined as $S \circ R := \{\langle a, c \rangle \in (A \times C) \mid \exists b \in B. \langle a, b \rangle \in R \wedge \langle b, c \rangle \in S\}$
- ▷ **Example 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 a 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 $<$, 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 $< := \{\langle a, b \rangle \mid (a \preceq b) \wedge a \neq b\}$ is a strict partial order. Conversely, if $<$ is a strict partial order, then $\preceq := \{\langle a, b \rangle \mid (a < b) \vee a = b\}$ 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** $f: X \rightarrow Y; x \mapsto y$ if $\langle x, y \rangle \in f$ (arrow notation)
 - ▷ call X the **domain** (write $\text{dom}(f)$), and Y the **codomain** ($\text{codom}(f)$) (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 $f: X \rightarrow Y$ **undefined at $x \in X$** , iff $\langle x, y \rangle \notin f$ for all $y \in Y$. (write $f(x) = \perp$)
- ▷ **Definition 3.6.19** If $f: X \rightarrow Y$ is a total relation, we call f a **total function** and write $f: X \rightarrow Y$. ($\forall x \in X. \exists^1 y \in Y. \langle x, y \rangle \in f$)
- ▷ **Notation 3.6.20** $f: x \mapsto y$ if $\langle x, y \rangle \in f$ (arrow notation)

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

⚠: 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 $A \rightarrow B$ ($A \rightarrow B$) 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

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

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

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



Lambda-Notation for Functions

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

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

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

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

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

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

this case we have pairs as arguments.

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



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 $f: S \rightarrow T$ 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 $\nu: \mathbb{N}_1 \rightarrow \mathbb{N}$ 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 $f: A \rightarrow \{n \in \mathbb{N} \mid n < \#(A)\}$.

▷ **Definition 3.6.32** We say that a set A is **countably infinite**, iff there is a bijective function $f: A \rightarrow \mathbb{N}$. A set is called **countable**, iff it is finite or countably infinite.

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

$$\triangleright \#(\{a, b, c\}) = 3 \qquad \text{(e.g. choose } f = \{\langle a, 0 \rangle, \langle b, 1 \rangle, \langle c, 2 \rangle\})$$

$$\triangleright \#(A \cup B) \leq \#(A) + \#(B)$$

$$\triangleright \#(A \cap B) \leq \min(\#(A), \#(B))$$

$$\triangleright \#(A \times B) = \#(A) \cdot \#(B)$$

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



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

Operations on Functions

\triangleright **Definition 3.6.34** If $f \in A \rightarrow B$ and $g \in B \rightarrow C$ are functions, then we call

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

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

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

\triangleright **Definition 3.6.36** Let $f: A \rightarrow B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call

$$\triangleright f(A') := \{b \in B \mid \exists a \in A'. \langle a, b \rangle \in f\} \text{ the } \mathbf{image} \text{ of } A' \text{ under } f,$$

$$\triangleright \mathbf{Im}(f) := f(A) \text{ the } \mathbf{image} \text{ of } f, \text{ and}$$

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




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

3.7 Standard ML: Functions as First-Class Objects

Enough theory, let us start computing with functions

We will use Standard ML for now


©: Michael Kohlhase
60


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 prover¹ — 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 $A \rightarrow B$, 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 $\mathcal{P} \rightarrow \mathcal{P}$, where \mathcal{P} is the type of proof data structures. Thus, using ML as a meta-language guaranteed that theorem prover could only construct valid proofs.

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

Standard ML (SML)

▷ Why this programming language?

¹The “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!



©: Michael Kohlhase

61



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 type** `s` `int`, `real`, `bool`, `string`, ...
 - ▷ **basic type constructor** `s` `->`, `*`,

- ▷ **basic operator** s numbers, true, false, +, *, -, >, ^, ... (⚠ overloading)
- ▷ **control structure** s if Φ then E_1 else E_2 ;
- ▷ **comment** s (**this is a comment **)



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

Definition 3.7.2 **type** s 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 declaration** s `val pi = 3.1415`;
 - ▷ **type declaration** s `type twovec = int * int`;
 - ▷ **function declaration** s `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 type** s 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 function** s, 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` $\hat{=}$ empty list, `::` $\hat{=}$ 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:

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

▷ **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

$$\text{diverges}(1) \rightsquigarrow 1 + \text{diverges}(2) \rightsquigarrow 1 + (2 + \text{diverges}(3)) \rightsquigarrow \dots$$


©: Michael Kohlhase

67



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"]
```



©: Michael Kohlhase

68



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 SML's basic types: strings.

We will now look at the `string` type of SML and how to deal with it. But before we do, let us recap what strings are. **String**s 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)



©: Michael Kohlhase

69



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;
```



©: Michael Kohlhase

70



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** $+$: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ 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** $+: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ withn $+(3)(2)$ instead of $+(⟨3, 2⟩)$.

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

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



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

▷ Cartesian and Cascaded Procedures (Brackets)

▷ **Definition 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)

▷ $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightsquigarrow \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ (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 **start value**

▷ **Example 3.7.17** Folding the iterator `op+` with start value 0:

```

foldl op+ 0 [x1,x2,x3] = x3+(x2+(x1+0))
    
```

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

©: Michael Kohlhase
73

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 [x1,x2,x3]
= x3 :: (x2 :: (x1 :: nil))
    
```

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

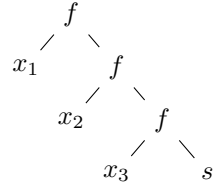
©: Michael Kohlhase
74

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)))
```

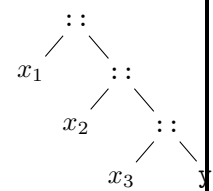


A tree diagram showing the evaluation of `foldr f s [x1, x2, x3]`. The root node is `f`, which takes `x1` and another `f` node as input. This second `f` node takes `x2` and a third `f` node as input. The third `f` node takes `x3` and `s` as input.


▷ **Example 3.7.20 (Appending Lists)**

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

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




A tree diagram showing the evaluation of `foldr op :: ys [x1, x2, x3]`. The root node is `::`, which takes `x1` and another `::` node as input. This second `::` node takes `x2` and a third `::` node as input. The third `::` node takes `x3` and `ys` as input.



©: Michael Kohlhase

75



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)



©: Michael Kohlhase

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:** $+: \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$ 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** $+: \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$ 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 $s: \mathbb{N}_1 \rightarrow \mathbb{N}_1$
- ▷ **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 $\mu(l, s(r)) = +(\mu(l, r), l)$
- ▷ **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 **constructor** $s \ c_i: S^{k_i} \rightarrow S$ with $k_i \geq 0$, such that
 - ▷ if $s_i \in S$ for all $1 \leq i \leq k_i$, then $c_i(s_1, \dots, s_{k_i}) \in S$ (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 **P1** and **P2**
 - P.2** Internal confusion is prevented **P4**
 - P.3** Inter-constructor confusion is averted by **P3**
 - P.4** Junk is prohibited by **P5**. □



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: $[1, 2, 3], [7, 7], [], \dots$
 - ▷ 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: $\text{nil} \in \mathcal{L}[\mathbb{N}]$ and $\text{cons}: \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \rightarrow \mathcal{L}[\mathbb{N}]$
- ▷ **Example 3.8.8** e.g. $[3, 2, 1] \hat{=} \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{nil})))$ and $[] \hat{=} \text{nil}$
- ▷ **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)** $\text{nil} \in \mathcal{L}[\mathbb{N}]$ (generation axiom (nil))
- ▷ **Axiom 3.8.11 (LP2)** $\text{cons}: \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \rightarrow \mathcal{L}[\mathbb{N}]$ (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 $\text{cons}(n, l)$ has property P
 then every list $l \in \mathcal{L}[\mathbb{N}]$ has property P .



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

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

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

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

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

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** $@: \mathcal{L}[\mathbb{N}] \times \mathcal{L}[\mathbb{N}] \rightarrow \mathcal{L}[\mathbb{N}]$ concatenates lists

Defining equations: $nil@l = l$ and $cons(n, l)@r = cons(n, l@r)$

▷ **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: $l = nil$: must have $s = r$.

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

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

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

□

▷ **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 **P** i there, we have used an axiom **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** $\lambda(nil) = o$ and $\lambda(cons(n, l)) = s(\lambda(l))$

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



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 types)**

```
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



©: Michael Kohlhase

85

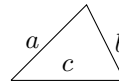


Somewhat surprisingly, finite enumeration types that are separate constructs in most programming languages are a special case of `datatype` declarations in SML. They are modeled by sets of base constructors, without any functional ones, so the base cases form the finite possibilities in this type. Note 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

```



©: Michael Kohlhase

86



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.


What's next?

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



©: Michael Kohlhase

88



JACOBS
UNIVERSITY

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 simple². 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 and terms to model SML types and expressions and programs.

Recall the diagram of things we want to model from Section 1.0 (see Figure 3.1). For representing data, we will use “ground constructor terms”, for algorithms we will make the idea of “defining equations” precise by defining “abstract procedures”, and for the machines, we will build an “abstract interpreter”. In all of these notions, we will employ abstract/mathematical methods to model and understand the relevant concepts.

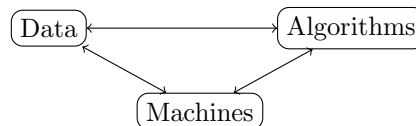


Figure 3.1: Data, Algorithms, and Machines

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.

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.

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).

Abstract Data Types (ADT)

- ▷ **Definition 3.10.1** Let $\mathcal{S}^0 := \{\mathbb{A}_1, \dots, \mathbb{A}_n\}$ be a finite set of symbols, then we call the set \mathcal{S} the set of **sorts** over the set \mathcal{S}^0 , if
 - ▷ $\mathcal{S}^0 \subseteq \mathcal{S}$ (base sort s are sorts)
 - ▷ If $\mathbb{A}, \mathbb{B} \in \mathcal{S}$, then $(\mathbb{A} \times \mathbb{B}) \in \mathcal{S}$ (product sort s are sorts)
 - ▷ If $\mathbb{A} \in \mathcal{X}$ and $\mathbb{B} \in \mathcal{S}^0$, then $(\mathbb{A} \rightarrow \mathbb{B}) \in \mathcal{S}$ (function sort s are sorts)
- ▷ **Definition 3.10.2** If c is a symbol and $\mathbb{A} \in \mathcal{S}$, then we call a pair $[c: \mathbb{A}]$ a **constructor declaration** for c over \mathcal{S} .
- ▷ **Definition 3.10.3** Let \mathcal{S}^0 be a set of symbols and \mathcal{D} a set of constructor declarations over \mathcal{S} , then we call the pair $\langle \mathcal{S}^0, \mathcal{D} \rangle$ an **abstract data type**
- ▷ **Example 3.10.4** $\langle \{\mathbb{B}\}, \{[T: \mathbb{B}], [F: \mathbb{B}]\} \rangle$ is an abstract data for truth values.
- ▷ **Example 3.10.5** $\langle \{\mathbb{N}\}, \{[o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}]\} \rangle$ represents unary natural numbers.
- ▷ **Example 3.10.6** $\langle \{\mathbb{N}, \mathcal{L}(\mathbb{N})\}, \{[o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}], [\text{nil}: \mathcal{L}(\mathbb{N})], [\text{cons}: \mathbb{N} \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})]\} \rangle$
In particular, the term $\text{cons}(s(o), \text{cons}(o, \text{nil}))$ represents the list $[1, 0]$
- ▷ **Example 3.10.7** $\langle \{\mathcal{S}^0, \mathcal{S}\}, \{[\iota: \mathcal{S}^0 \rightarrow \mathcal{S}], [\rightarrow: \mathcal{S} \times \mathcal{S}^0 \rightarrow \mathcal{S}], [\times: \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}]\} \rangle$
(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 sorts \mathcal{S}^0 and \mathcal{S} , which are just abstract mathematical objects that model the sets of base sorts and sorts from Definition 3.10.1 – without being the same, even though the symbols look the same. The declaration $[\iota: \mathcal{S}^0 \rightarrow \mathcal{S}]$ introduces an inclusion of the base sorts into the sorts: if a is a base sort, then $\iota(a)$ is a sort – this is a standard trick to represent the subset relation via a constructor.

Finally the two remaining declarations $[\rightarrow: \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}]$ and $[\times: \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}]$ 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 $[t: \mathbb{T}] \in \mathcal{D}$, 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 $[c: \mathbb{A} \rightarrow \mathbb{T}] \in \mathcal{D}$.

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

▷ **Definition 3.10.9** If $t = c(t')$ then we say that the symbol c is the **head** of t (write **head**(t)). If $t = a$, then **head**(t) = a ; **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 $\text{cons}(s(s(o)), \text{cons}(s(o), \text{nil}))$ 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 (inductively defined) 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 $\mathcal{B} := \langle \{\mathbb{B}\}, \{[T: \mathbb{B}], [F: \mathbb{B}]\} \rangle$ 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.

$$\begin{aligned} \neg: & \langle \neg: \mathbb{B} \rightarrow \mathbb{B}; \{\neg(T) \rightsquigarrow F, \neg(F) \rightsquigarrow T\} \rangle, \\ \wedge: & \langle \wedge: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}; \{\wedge(T, T) \rightsquigarrow T, \wedge(T, F) \rightsquigarrow F, \wedge(F, T) \rightsquigarrow F, \wedge(F, F) \rightsquigarrow F\} \rangle, \\ \vee: & \langle \vee: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}; \{\vee(T, T) \rightsquigarrow T, \vee(T, F) \rightsquigarrow T, \vee(F, T) \rightsquigarrow T, \vee(F, F) \rightsquigarrow F\} \rangle. \end{aligned}$$

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 (12r) and use them as rules.
- ▷ **Definition 3.11.1** Let \mathcal{A} be an abstract data type and $s, t \in \mathcal{T}_{\mathbb{T}}^g(\mathcal{A})$ ground constructor terms over \mathcal{A} , then we call a pair $s \rightsquigarrow t$ a **rule** for f , if $\text{head}(s) = f$.

- ▷ **Example 3.11.2** turn $\lambda(\text{nil}) = o$ and $\lambda(\text{cons}(n, l)) = s(\lambda(l))$
to $\lambda(\text{nil}) \rightsquigarrow o$ and $\lambda(\text{cons}(n, l)) \rightsquigarrow s(\lambda(l))$
- ▷ **Definition 3.11.3** Let $\mathcal{A} := \langle S^0, \mathcal{D} \rangle$, then call a quadruple $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ 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 $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$.
- ▷ **Definition 3.11.4** A **computation** of an abstract procedure p is a sequence of ground constructor terms $t_1 \rightsquigarrow t_2 \rightsquigarrow \dots$ 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

- ▷ **Example 3.11.7** Consider the abstract procedures

$$\langle \rho::\mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{\rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil})), \rho(\text{nil}) \rightsquigarrow \text{nil}\} \rangle$$

$$\langle @::\mathcal{L}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{@(\text{cons}(n, l), r) \rightsquigarrow \text{cons}(n, @(l, r)), @(\text{nil}, l) \rightsquigarrow l\} \rangle$$

- ▷ Then we have the following abstract computation

- ▷ $\rho(\text{cons}(2, \text{cons}(1, \text{nil}))) \rightsquigarrow @(\rho(\text{cons}(1, \text{nil})), \text{cons}(2, \text{nil}))$ ($\rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil}))$ with $n = 2$ and $l = \text{cons}(1, \text{nil})$)
- ▷ $@(\rho(\text{cons}(1, \text{nil})), \text{cons}(2, \text{nil})) \rightsquigarrow @(@(\rho(\text{nil}), \text{cons}(1, \text{nil})), \text{cons}(2, \text{nil}))$ ($\rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil}))$ with $n = 1$ and $l = \text{nil}$)
- ▷ $@(@(\rho(\text{nil}), \text{cons}(1, \text{nil})), \text{cons}(2, \text{nil})) \rightsquigarrow @(@(\text{nil}, \text{cons}(1, \text{nil})), \text{cons}(2, \text{nil}))$ ($\rho(\text{nil}) \rightsquigarrow \text{nil}$)
- ▷ $@(@(\text{nil}, \text{cons}(1, \text{nil})), \text{cons}(2, \text{nil})) \rightsquigarrow @(\text{cons}(1, \text{nil}), \text{cons}(2, \text{nil}))$ ($@(\text{nil}, l) \rightsquigarrow l$ with $l = \text{cons}(1, \text{nil})$)
- ▷ $@(\text{cons}(1, \text{nil}), \text{cons}(2, \text{nil})) \rightsquigarrow \text{cons}(1, @(\text{nil}, \text{cons}(2, \text{nil})))$ ($@(\text{cons}(n, l), r) \rightsquigarrow \text{cons}(n, @(l, r))$ with $n = 1$, $l = \text{nil}$, and $r = \text{cons}(2, \text{nil})$)
- ▷ $\text{cons}(1, @(\text{nil}, \text{cons}(2, \text{nil}))) \rightsquigarrow \text{cons}(1, \text{cons}(2, \text{nil}))$ ($@(\text{nil}, l) \rightsquigarrow l$ with $l = \text{cons}(2, \text{nil})$)

Aha: ρ terminates on the argument $\text{cons}(2, \text{cons}(1, \text{nil}))$



In the example above we try to understand the mechanics of “replacing equals by equals” with

two concrete abstract procedures; one for ρ (think “reverse”) and one for $@$ (think “append”). We start with an initial term $\rho(\text{cons}(2, \text{cons}(1, \text{nil})))$ and realize that we can make this look equal to the left-hand side of the first rule of the abstract procedure ρ by making n be 2 and l be $\text{cons}(1, \text{nil})$. As the left hand side is equal to our expression, we can replace it with the right hand side of the same rule, again with $n = 2$ and $l = \text{cons}(1, \text{nil})$. This gives us the first computation step.

The second computation step goes similarly: we work on the result of the first step and again find a left-hand-side of a rule that we can make equal. But this time we do not match the *whole* expression and replace it with the right-hand-side, but only a part. We have indicated what we are working on by making the procedure symbol red.

The third computation step proceeds in style and eliminates the last occurrence of the procedure symbol ρ via the second rule. The last three steps proceed in kind and eliminate the occurrences of the procedure symbol $@$, so that we are left with a ground constructor term – that does not contain procedure symbols. This term is the last in the computation and thus constitutes its result.

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 3.11.8 (Idea)** Replace equals by equals! (this is licensed by the rules)

- ▷ **Input:** an abstract procedure $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ and an **argument** $a \in \mathcal{T}_{\mathbb{A}}^g(\mathcal{A})$.
- ▷ **Output:** a **result** $r \in \mathcal{T}_{\mathbb{R}}^g(\mathcal{A})$.
- ▷ **Process:**
 - ▷ **find a part** $t := f(t_1, \dots, t_n)$ 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 3.11.9** We say that an abstract procedure $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ **terminates** (on $a \in \mathcal{T}_{\mathbb{A}}^g(\mathcal{A})$), 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 yet have the means to write down rules: they contain variables, which are not allowed in ground constructor rules. So 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 3.11.10** 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. $x_{\mathbb{A}}, n_{\mathbb{N}_1}, x_{\mathbb{C}^3}, \dots$
- ▷ **Definition 3.11.11** 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 3.11.12** If t is a constructor term, then we denote the set of variables occurring in t with $\text{free}(t)$. If $\text{free}(t) = \emptyset$, 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 3.11.13** 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 $[t: \mathbb{T}] \in \mathcal{D}$, 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 $[c: \mathbb{A} \rightarrow \mathbb{T}] \in \mathcal{D}$.

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 3.11.14** We define the **depth** of a term t by the way it is constructed.

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



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 $\sigma \in \mathcal{V} \rightarrow \mathcal{T}(\mathcal{A}; \mathcal{V})$, then we call σ a **substitution** on \mathcal{A} , iff $\text{supp}(\sigma) := \{x_{\mathbb{A}} \in \mathcal{V}_{\mathbb{A}} \mid \sigma(x_{\mathbb{A}}) \neq x_{\mathbb{A}}\}$ is finite and $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$. $\text{supp}(\sigma)$ is called the **support** of σ .
- ▷ **Notation 3.12.2** We denote the substitution σ with $\text{supp}(\sigma) = \{x_{\mathbb{A}_i}^i \mid 1 \leq i \leq n\}$ and $\sigma(x_{\mathbb{A}_i}^i) = t_i$ by $[t_1/x_{\mathbb{A}_1}^1], \dots, [t_n/x_{\mathbb{A}_n}^n]$.
- ▷ **Definition 3.12.3 (Substitution Extension)** Let σ be a substitution, then we denote with $\sigma, [t/x_{\mathbb{A}}]$ the function $\{(y_{\mathbb{B}}, t) \in \sigma \mid y_{\mathbb{B}} \neq x_{\mathbb{A}}\} \cup \{(x_{\mathbb{A}}, t)\}$.
($\sigma, [t/x_{\mathbb{A}}]$ 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.



Note that substitutions are “well-sorted” since we assume that $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ in Definition 3.12.1. In other words, a substitution may only assign terms of the sort, the variable prescribes.

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 $\sigma: \mathcal{V} \rightarrow \mathcal{T}(\mathcal{A}; \mathcal{V})$ to a function $\sigma: \mathcal{T}(\mathcal{A}; \mathcal{V}) \rightarrow \mathcal{T}(\mathcal{A}; \mathcal{V})$.
- ▷ **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 . We also say that t is an instance of s .

▷ **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 elided here)

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

▷ $\sigma(x_{\mathbb{A}}) = t$ if $[t/x_{\mathbb{A}}] \in \sigma$.

▷ $\sigma(\langle a, b \rangle) = \langle \sigma(a), \sigma(b) \rangle$.

▷ $\sigma(f(a)) = f(\sigma(a))$.

▷ this definition uses the inductive structure of the terms.



Note that even though a substitution in and of itself is a total function on variables, it can be *extended* to a function on constructor terms by Definition 3.12.7. But we do not have a notation for this function. In particular, we may not write $[t/s]$, unless s is 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 $\text{dp}(t)$ using Definition 3.11.13 and Definition 3.12.4

P.1 By Definition 3.11.13 we have to consider four cases

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

P.1.2 $t \in \mathcal{V}_{\mathbb{T}}$: We have $\sigma(t) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$ by Definition 3.12.1,

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 $[c: \mathbb{A} \rightarrow \mathbb{T}] \in \mathcal{D}$: 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 $(c(\sigma(a))) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$. \square



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** $\rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil}))$. (ρ 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 $\mathbb{A} \in \mathcal{S}$ and let $f \notin \mathcal{D}$ be a symbol, then we call a pair $[f: \mathbb{A}]$ a **procedure declaration** for f over \mathcal{S} .
- ▷ **Definition 3.13.3** We call a finite set Σ of procedure declarations for distinct symbols a **signature** over \mathcal{A} .
- ▷ **Idea:** add the following rules to the definition of constructor terms
 - ▷ $\mathbb{T} \in \mathcal{S}^0$ and $[p: \mathbb{T}] \in \Sigma$, or
 - ▷ t is of the form $f(a)$, where a is a term of sort \mathbb{A} and there is a procedure declaration $[f: \mathbb{A} \rightarrow \mathbb{T}] \in \Sigma$.



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 3.13.4** 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 $[t: \mathbb{T}] \in \mathcal{D}$ or $[t: \mathbb{T}] \in \Sigma$, 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 $[c: \mathbb{A} \rightarrow \mathbb{T}] \in \mathcal{D}$ or a procedure declaration $[c: \mathbb{A} \rightarrow \mathbb{T}] \in \Sigma$.

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, which 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 3.13.5** 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 3.13.6** 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 3.13.7** $f(a)$ is a subterm of the terms $f(a)$ and $h(g(f(a), f(b)))$, and an immediate subterm of $h(f(a))$.



If we look at the definition of immediate subterms in Definition 3.13.5, then we can interpret the construction as reading the step cases in the rules for term construction (see Definition 3.13.4) backwards: What does it take to construct a term? The subterm relation corresponds to going back over the rules an arbitrary number of times (including zero times).

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 abstract data type, Σ a signature over \mathcal{A} , and $f \notin (\text{dom}(\mathcal{D}) \cup \text{dom}(\Sigma))$ a symbol, then we call $f(s) \rightsquigarrow r$ a **rule** for $[f: \mathbb{A} \rightarrow \mathbb{B}]$ over Σ , if $s \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma; \mathcal{V})$ has no duplicate variables, constructors, or defined functions and $r \in \mathcal{T}_{\mathbb{B}}(\mathcal{A}, \Sigma, [f: \mathbb{A} \rightarrow \mathbb{B}]; \mathcal{V})$.
- ▷ **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 $\mathcal{P} := \langle f: \mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ an **abstract procedure** over Σ , iff \mathcal{R} is a set of rules for $[f: \mathbb{A} \rightarrow \mathbb{R}] \in \Sigma$. We say that \mathcal{P} **induces** the procedure declaration $[f: \mathbb{A} \rightarrow \mathbb{R}]$.
- ▷ **Example 3.14.3** Let \mathcal{A} be the union of the abstract data types from Exam-

ple 3.10.6 and Example 3.10.14, then

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

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



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 computation in the interpreter).

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 $\mathcal{P} := \mathcal{P}_1, \dots, \mathcal{P}_n$ 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 $\Sigma = \Sigma', [f: \mathbb{A}]$, where
 - ▷ \mathcal{P}' is an abstract program over Σ'
 - ▷ and \mathcal{P}_n is an abstract procedure over Σ' that induces the procedure declaration $[f: \mathbb{A}]$.

▷ **Example 3.14.5** The two abstract procedures from Example 3.11.7

$$\langle @:: \mathcal{L}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{ @(\text{cons}(n, l), r) \rightsquigarrow \text{cons}(n, @(l, r)), @(nil, l) \rightsquigarrow l \} \rangle$$

$$\langle \rho:: \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{ \rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, nil)), \rho(nil) \rightsquigarrow nil \} \rangle$$

constitute an abstract program over the abstract data type from Example 3.10.6:

$$\langle \{ \mathbb{N}, \mathcal{L}(\mathbb{N}) \}, \{ [o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}], [nil: \mathcal{L}(\mathbb{N})], [cons: \mathbb{N} \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})] \} \rangle$$



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 $\mathcal{P} := \langle f :: \mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ **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 Definition 3.14.6 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 to 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 (final)

▷ **Definition 3.15.2 (Call-by-Value Interpreter)** Given an abstract program \mathcal{P} and a ground constructor term a , an **abstract call-by-value interpreter** creates a computation $a_1 \rightsquigarrow a_2 \rightsquigarrow \dots$ with $a = a_1$ 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 delay computing the values of their arguments. They are uniquely suited for dealing with objects that are potentially infinite in some form. “Infinite” arguments are not a problem, since “lazy” languages only compute enough of them so that the overall computation can proceed. 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`.

Even though the problem of termination is difficult in full generality, we can make some progress on it. 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** $\tau: \mathbb{N}_1 \rightarrow \mathbb{N}_1$, 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 $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ 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 the abstract procedure

$$\langle \sigma :: \mathbb{N}_1 \rightarrow \mathbb{N}_1 ; \{ \sigma(o) \rightsquigarrow o, \sigma(s(n)) \rightsquigarrow n + \sigma(n) \} \rangle$$

▷ **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.



Note that the procedure sketched in Example 3.15.4 is not really an abstract procedure since it has the even/odd condition, we cannot express in our setup. But even so, it shows us that termination is difficult.

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, \dots , and even though there is a huge literature in mathematics about this problem – the Collatz Conjecture, a proof that τ diverges on all arguments is still missing.

Another clue to the difficulty of the termination problem is (Theorem 14.5.10) that there cannot be a program that reliably tells of any program whether it will terminate.

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, \dots in \mathbb{A} , such that $\langle a_n, a_{n+1} \rangle \in R$ for all $n \in \mathbb{N}$.

▷ **Definition 3.15.9** We say that R **terminates (on $a \in \mathbb{A}$)**, iff there is no infinite chain in R (that begins with a).

▷ **Definition 3.15.10** \mathcal{P} **diverges (on $a \in \mathbb{A}$)**, iff it does not terminate on a .

▷ **Theorem 3.15.11** Let $\mathcal{P} = \langle f :: \mathbb{A} \rightarrow \mathbb{R} ; \mathcal{R} \rangle$ 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 3.15.12** Let $\mathcal{P} = \langle f :: \mathbb{A} \rightarrow \mathbb{R} ; \mathcal{R} \rangle$ be an abstract procedure, then we call the function $\{ \langle a, b \rangle \mid a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}, \Sigma ; \mathcal{V}) \text{ and } \mathcal{P} \text{ terminates for } a \text{ with } b \}$ in $\mathbb{A} \rightarrow \mathbb{B}$ the **result function** of \mathcal{P} .

▷ **Theorem 3.15.13** Let $\mathcal{P} = \langle f :: \mathbb{A} \rightarrow \mathbb{B} ; \mathcal{D} \rangle$ be a terminating abstract procedure, then its result function satisfies the equations in \mathcal{D} .



We should read Theorem 3.15.13 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.

We conclude the section with a reflection the role of abstract data, procedures, and machines vs. concrete ones (see Figure 3.1); the result of this is that the abstract model is actually simpler than the concrete ones, since it abstracts away from many nasty real-world restrictions.

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 $\sigma: \mathbb{N}_1 \rightarrow \mathbb{N}_1$ with $\sigma(o) \rightsquigarrow o, \sigma(s(n)) \rightsquigarrow n + \sigma(n)$
with $\sigma': \mathbb{N}_1 \rightarrow \mathbb{N}_1$ 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, ...
 - ▷ 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!)

©: Michael Kohlhase 112

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;

```



©: Michael Kohlhase

114



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 a 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.



©: Michael Kohlhase

115



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 its 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!



©: Michael Kohlhase

116



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` ($\hat{=}$ keyboard input and screen)
- ▷ **Example 4.2.1 (Input)** via `TextIO.inputLine : TextIO.instream -> string`

```

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

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

`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 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 instr = 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 be 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 handlers 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 $\langle c_1, \dots, c_n \rangle$ as " $c_1 \dots c_n$ ", for instance "**a, b, c**" for $\langle a, b, c \rangle$.
- ▷ **Example 5.1.4** Take $A = \{\mathfrak{h}, 1, / \}$ as an alphabet. Each of the symbols \mathfrak{h} , 1, and $/$ is a character. The vector $\langle /, /, 1, \mathfrak{h}, 1 \rangle$ 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** $\text{conc}(s, t)$ of two strings $s = \langle s_1, \dots, s_n \rangle \in A^n$ and $t = \langle t_1, \dots, t_m \rangle \in A^m$ is defined as $\langle s_1, \dots, s_n, t_1, \dots, t_m \rangle \in A^{n+m}$.

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



©: Michael Kohlhase

126



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 string** s and $A^* := A^+ \cup \{\epsilon\}$ of **string** s .
- ▷ **Example 5.1.8** If $A = \{a, b, c\}$, then $A^* = \{\epsilon, a, b, c, aa, ab, ac, ba, \dots, aaa, \dots\}$.
- ▷ **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** $\text{conc}(L_1, L_2)$ of two languages L_1 and L_2 over the same alphabet is defined as $\text{conc}(L_1, L_2) := \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$.



©: Michael Kohlhase

127



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 strings: 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** $\text{conc}(/, 1, h)$ is a substring of $\text{conc}(/, /, 1, h, 1)$, whereas $\text{conc}(/, 1, 1)$ is not.

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

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

▷ **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 $<_{\text{lex}}$ on A^* by

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

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

▷ **Theorem 5.1.19** $<_{\text{lex}}$ is a **strict partial order** on A^* . Moreover, if $<_A$ is **linear** on A , then $<_{\text{lex}}$ is **linear** on A^* .

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



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 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 $c: A \rightarrow B^+$ a **character code**. A string $c(a) \in \{c(a) \mid a \in A\}$ is called a **codeword**.

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

▷ **Example 5.2.3** Let $A = \{a, b, c\}$ and $B = \{0, 1\}$, then $c: A \rightarrow B^+$ 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) $c: A \rightarrow B^+$ to a function $c': A^* \rightarrow B^*$ is defined as $c'(\langle a_1, \dots, a_n \rangle) = \langle c(a_1), \dots, c(a_n) \rangle$.

▷ **Example 5.2.5** The extension c' of c from the above example on the string "b, b, a, b, c"

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

▷ **Definition 5.2.6** A (character) code $c: A \rightarrow B^+$ is a **prefix code** iff none of the codewords is a proper prefix to another codeword, i.e.,

$$\forall x, y \in A^* x \neq y \Rightarrow (c(x) \not\prec c(y) \wedge c(y) \not\prec c(x))$$



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 $\mu: \mathcal{R} \rightarrow \{., -\}$.




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. $\mu'(\text{AT}) = \mu'(\text{W})$. While this is mathematically

a problem for decoding texts encoded by Morse code, for humans (who were actually decoding it) this is 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 $c' : A^* \rightarrow B^*$ is called a **code on strings** or short **string code** if c' is an injective function.

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

▷ **Proof:** we give an example

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

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

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

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 $c' : A^* \rightarrow B^*$ of a prefix code $c : A \rightarrow B^+$ 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 $a = a_0, \dots, a_n$. And we only know $c'(a) = c(a_0), \dots, c(a_n)$.

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 $c(a_1), \dots, c(a_n)$, 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 a 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 $c: A \rightarrow B^+$ be a code and $*$ $\notin B$ be a character, then there is a prefix code $c^*: A \rightarrow (B \cup \{*\})^+$, 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

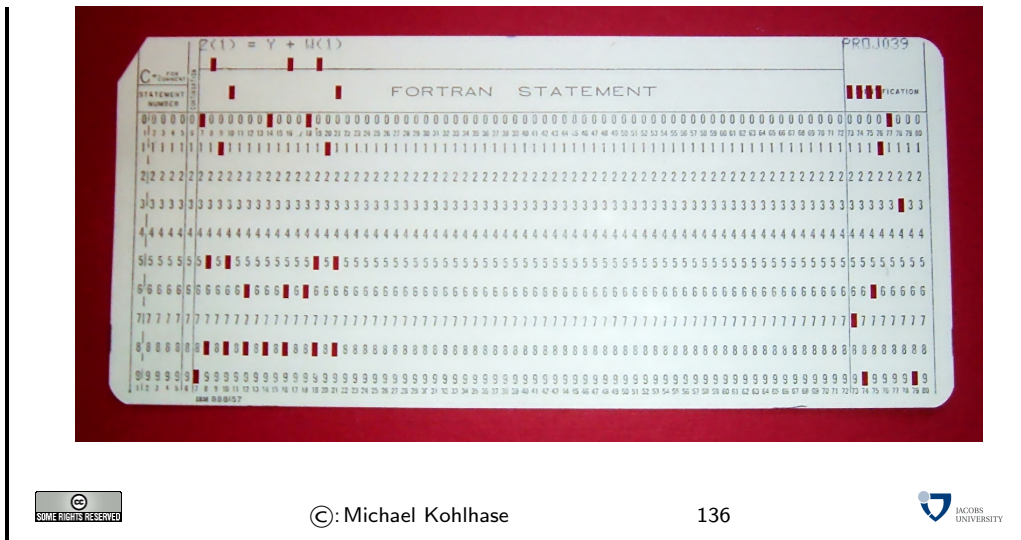
- ▷ **Motivated by punchcards:** The character 0 (binary 0000000) carries no information NUL, (used as dividers)
Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)
- ▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)



Punch cards were the 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+00FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000 – U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- ▷ **Example 5.3.12** \$ = U+0024 is encoded as 00100100 (1 byte)
- ç = U+00A2 is encoded as 11000010,10100010 (two bytes)
- € = 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 integer division)
- ▷ E_{un} is based on the alphabet $\Sigma_{\text{un}} := C_{\text{un}} \cup V \cup F_{\text{un}}^2 \cup B$, where
 - ▷ $C_{\text{un}} := \{/ \}^*$ is a set of **constant name** s,
 - ▷ $V := \{x\} \times \{1, \dots, 9\} \times \{0, \dots, 9\}^*$ is a set of **variable name** s,
 - ▷ $F_{\text{un}}^2 := \{\text{add, sub, mul, div, mod}\}$ is a set of (binary) **function name** s, and
 - ▷ $B := \{(\cdot)\} \cup \{,\}$ is a set of **structural character** s. (\triangleleft “”, “(”, “)”, “,” characters!)
- ▷ define strings in stages: $E_{\text{un}} := \bigcup_{i \in \mathbb{N}} E_{\text{un}}^i$, where
 - ▷ $E_{\text{un}}^1 := C_{\text{un}} \cup V$
 - ▷ $E_{\text{un}}^{i+1} := \{a, \text{add}(a,b), \text{sub}(a,b), \text{mul}(a,b), \text{div}(a,b), \text{mod}(a,b) \mid a, b \in E_{\text{un}}^i\}$

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



©:Michael Kohlhase

140



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

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

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

A formal Language for Unary Arithmetics (Examples)

▷ **Example 5.4.1** $\text{add}(//////, \text{mul}(x1902, ///)) \in E_{\text{un}}$

▷ **Proof:** we proceed according to the definition

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

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

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

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

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

▷ other examples:

▷ $\text{div}(x201, \text{add}(///, x12))$

▷ $\text{sub}(\text{mul}(///, \text{div}(x23, ///)), ///)$

▷ what does it all mean? (nothing, E_{un} is just a set of strings!)



©:Michael Kohlhase

141



To show that a string is an expression s of unary arithmetics, we have to show that it is in the formal language E_{un} . As E_{un} is the union over all the E_{un}^i , the string s must already be a member of a set E_{un}^j for some $j \in \mathbb{N}$. So we reason by the definition establishing 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. $\text{add}(\text{////////}, \text{mul}(\text{x1902}, \text{////})) \mapsto 6 + (x_{1902} \cdot 3)$ (but what does this mean?)

▷ looks like we have to give a meaning to the variables as well, e.g. $x_{1902} \mapsto 3$, then $\text{add}(\text{////////}, \text{mul}(\text{x1902}, \text{////})) \mapsto 6 + (3 \cdot 3) = 15$



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 humans¹ 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 $\mathcal{A} := C_{\text{bool}} \cup V \cup F_{\text{bool}}^1 \cup F_{\text{bool}}^2 \cup B$, where $C_{\text{bool}} = \{0, 1\}$, $F_{\text{bool}}^1 = \{-\}$ and $F_{\text{bool}}^2 = \{+, *\}$. (V and B as in E_{un})

▷ **Definition 6.1.2 (Formal Language)** $E_{\text{bool}} := \bigcup_{i \in \mathbb{N}} E_{\text{bool}}^i$, where $E_{\text{bool}}^1 := C_{\text{bool}} \cup V$ and $E_{\text{bool}}^{i+1} := \{a, (-a), (a+b), (a*b) \mid a, b \in E_{\text{bool}}^i\}$.

▷ **Definition 6.1.3** Let $a \in E_{\text{bool}}$. The minimal i , such that $a \in E_{\text{bool}}^i$ is called the **depth** of a .

▷ $e_1 := ((-x1)+x3)$ (depth 3)

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

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



©: Michael Kohlhase

143



Boolean Expressions as Structured Objects.

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

¹until 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.

of them as constructor terms with variables

▷ **Definition 6.1.4** The abstract data type

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

▷ via the translation


▷ **Definition 6.1.5** $\sigma: E_{\text{bool}} \rightarrow \mathcal{T}_{\mathbb{B}}(\mathcal{B}; \mathcal{V})$ defined by

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

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

▷ : 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 $\mathcal{I}(C_{\text{bool}}) \subseteq \mathcal{U}$, $\mathcal{I}(F_{\text{bool}}^1) \subseteq \mathcal{F}(\mathcal{U}; \mathcal{U})$, and $\mathcal{I}(F_{\text{bool}}^2) \subseteq \mathcal{F}(\mathcal{U}^2; \mathcal{U})$.

▷ **Definition 6.1.9** A function $\varphi: V \rightarrow \mathcal{U}$ 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** $\mathcal{I}_{\varphi}: E_{\text{bool}} \rightarrow \mathcal{U}$ is defined recursively: Let $c \in C_{\text{bool}}$, $a, b \in E_{\text{bool}}$, and $x \in V$, then

$$\triangleright \mathcal{I}_{\varphi}(c) = \mathcal{I}(c), \text{ for } c \in C_{\text{bool}}$$

$$\triangleright \mathcal{I}_{\varphi}(x) = \varphi(x), \text{ for } x \in V$$

- ▷ $\mathcal{I}_\varphi(\bar{a}) = \mathcal{I}(-)(\mathcal{I}_\varphi(a))$
- ▷ $\mathcal{I}_\varphi(a + b) = \mathcal{I}(+)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$ and $\mathcal{I}_\varphi(a * b) = \mathcal{I}(*)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$
- ▷ $\mathcal{U} = \{\mathbf{T}, \mathbf{F}\}$ with $0 \mapsto \mathbf{F}, 1 \mapsto \mathbf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg$.
- ▷ $\mathcal{U} = E_{\text{un}}$ with $0 \mapsto /, 1 \mapsto //, + \mapsto \text{div}, * \mapsto \text{mod}, - \mapsto \lambda x.5$.
- ▷ $\mathcal{U} = \{0, 1\}$ with $0 \mapsto 0, 1 \mapsto 1, + \mapsto \min, * \mapsto \max, - \mapsto \lambda x.1 - x$.



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

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

Evaluating Boolean Expressions

▷ **Example 6.1.11** Let $\varphi := [\mathbf{T}/x_1], [\mathbf{F}/x_2], [\mathbf{T}/x_3], [\mathbf{F}/x_4]$, and $\mathcal{I} = \{0 \mapsto \mathbf{F}, 1 \mapsto \mathbf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\}$, then

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

▷ What a mess!



Boolean Algebra

▷ **Definition 6.1.12** A **Boolean algebra** is E_{bool} together with the models

- ▷ $\langle \{\mathbf{T}, \mathbf{F}\}, \{0 \mapsto \mathbf{F}, 1 \mapsto \mathbf{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\} \rangle$.
- ▷ $\langle \{0, 1\}, \{0 \mapsto 0, 1 \mapsto 1, + \mapsto \max, * \mapsto \min, - \mapsto \lambda x.1 - x\} \rangle$.

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

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

▷ **Definition 6.1.14** We call two expressions $e_1, e_2 \in E_{\text{bool}}$ **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 $\mathcal{I}_\varphi((\overline{e_1} + e_2) * (e_1 + \overline{e_2})) = \text{T}$ 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:

¬		
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}	$e_1 := x_{123} * x_{72}$	$e_2 := \overline{x_{123}}$	$e_3 := x_{123} * x_4$	$e_3 + e_2$
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 $a, b, c \in E_{\text{bool}}$, $\circ \in \{+, *\}$, 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 Chapter 10.

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 $f, g: \mathbb{B}^n \rightarrow \mathbb{B}$ 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 $\{X\} \times \{1, \dots, 9\}^+ \times \{0, \dots, 9\}^*$)
- ▷ **Definition 6.2.3** Let $e \in E_{\text{bool}}$ and $\{x_1, \dots, x_n\}$ the set of variables in e , then call $VL(e) := \langle x_1, \dots, x_n \rangle$ the **variable list** of e , iff $x_i <_{\text{lex}} x_j$ where $i \leq j$.
- ▷ **Definition 6.2.4** Let $e \in E_{\text{bool}}$ with $VL(e) = \langle x_1, \dots, x_n \rangle$, then we call the function

$$f_e: \mathbb{B}^n \rightarrow \mathbb{B} \text{ with } f_e: c \mapsto \mathcal{I}_{\varphi_c}(e)$$
 the Boolean function **induced** by e , where $\varphi_{\langle c_1, \dots, c_n \rangle}: x_i \mapsto c_i$. 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	$f_{x_1 * (\bar{x}_2 + x_3)}$
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)



©:Michael Kohlhase

151



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

Boolean Polynomials

▷ special form Boolean Expressions

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

For literals x_i , write x_i^1 , for \bar{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)



©:Michael Kohlhase

152



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 $f: \mathbb{B}^n \rightarrow \mathbb{B}$ 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 normal form (DNF)** of f is $\sum_{c \in f^{-1}(1)} M_c$ (also called the **canonical sum** (written as $\text{DNF}(f)$))

▷ **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 $\text{CNF}(f)$))

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 $e \in E_{\text{bool}}$ is the number of operators in e .

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

▷ **Definition 6.2.12** Let $f: \mathbb{B}^n \rightarrow \mathbb{B}$ 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

We have learned that we can always realize a Boolean function as a Boolean Expression – which can ultimately serve as a blueprint for a realization in a circuit. But to get a full understanding of the problem, we should be able to predict the cost/size of the realizing Boolean expressions.

In principle, knowing how to realize a Boolean function would be sufficient to start a business as a chip manufacturer: clients come with a Boolean function, and the manufacturer computes the Boolean expression and produces the chip. But to be able to quote a price for the chip, the manufacturer needs to be able to predict the size of the Boolean expression *before* it is actually computed. Thus the need for prediction.

Before we turn to the prediction of cost/size for Boolean expressions, we will introduce the basic tools for this kind of analysis: A way of even expressing the results.

6.3.1 The Mathematics of Complexity

We will now introduce some very basic concepts of computational complexity theory – the discipline of classifying algorithms and the computational problems they solve into classes of inherent difficulty. In this classification, we are interested only in properties that are inherent to the problem or algorithm, not in the specific hardware or realization. This requires a special vocabulary which we will now introduce.

We want to classify the resource consumption of a computational process or problem in terms of the size of the “size” of the problem description. Thus we use functions from \mathbb{N} (the problem size) to \mathbb{N} (the resources consumed).

Example 6.3.1 (Mazes) We have already looked at different algorithms for constructing mazes in Example 2.2.14. There the problem size was the number of cells in the maze and the resource was the runtime in micro-seconds. The simple computation there shows that different growth patterns have great effects on computation time.

Note that the actual resource consumption function of actual computations depends very much on external factors that have nothing to do with the algorithm itself, e.g. the startup time or memory space of the runtime environment, garbage collection cycles, memory size that forces swapping, etc, and of course the processor speed itself. Therefore we want to abstract away from such factors in disregarding constant factors (processor speed) and startup effects.

Finally, we are always interested in the “worst-case behavior” – i.e. the maximal resource consumption over all possible inputs for the algorithm. But – as this may be difficult to predict, we are interested in upper and lower bounds of that.

All of these considerations lead to the notion of Landau sets we introduce now

The Landau Notations (aka. “big-O” Notation)

▷ **Definition 6.3.2** Let $f, g: \mathbb{N} \rightarrow \mathbb{N}$, 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 6.3.3** The three **Landau sets** $O(g), \Omega(g), \Theta(g)$ are defined as

$$\triangleright O(g) = \{f \mid \exists k > 0. f \leq_a k \cdot g\}$$

$$\triangleright \Omega(g) = \{f \mid \exists k > 0. f \geq_a k \cdot g\}$$

$$\triangleright \Theta(g) = O(g) \cap \Omega(g)$$

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 ”)

▷ **Notation 6.3.4** (\triangleleft) We often see $f = O(g)$ as a statement of complexity; this is a funny notation for $n \in fO(g)$!

The practical power of Landau sets as a tool for classifying resource consumption of algorithms and computational problems comes from the way we can calculate with them.

Computing with Landau Sets

▷ **Lemma 6.3.5** We have the following computation rules for Landau sets:

- ▷ If $k \neq 0$ and $f \in O(g)$, then $(kf) \in O(g)$.
- ▷ If $f_i \in O(g_i)$, then $(f_1 + f_2) \in O(g_1 + g_2)$
- ▷ If $f_i \in O(g_i)$, then $(f_1 f_2) \in O(g_1 g_2)$

▷ **Notation 6.3.6** If e is an expression in n , we write $O(e)$ for $O(\lambda n.e)$ (for Ω/Θ too)

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

▷ **Example 6.3.7** $(\lambda n.263748) \in O(1)$

▷ **Example 6.3.8** $(\lambda n.26n + 372) \in O(n)$

▷ **Example 6.3.9** $(\lambda n.857n^{10} + 7342n^7 + 26n^2 + 902) \in O(n^{10})$

▷ **Example 6.3.10** $(\lambda n.3 \cdot 2^n + 72) \in O(2^n)$

▷ **Example 6.3.11** $(\lambda n.3 \cdot 2^n + 7342n^7 + 26n^2 + 722) \in O(2^n)$



Note that the “arithmetic operations” in Lemma 6.3.5 are applied to functions as “pointwise” operations.

A useful consequence of the addition rule is the following:

Lemma 6.3.12 If $f_i \in O(g)$, then $(f_1 + f_2) \in O(g)$

The examples show that the computation rules from Lemma 6.3.5 allow us to classify complex functions by dropping lower-growth summands and scalar factors. They also show us that we only need to consider very simple representatives for the Landau sets.

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 6.3.13** 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.



With the basics of complexity theory well-understood, we can now analyze the cost-complexity of Boolean expressions that realize Boolean functions.

6.3.2 Asymptotic Bounds for Costs of Boolean Expressions

In this subsection we will derive some results on the (cost)-complexity of realizing a Boolean function as a Boolean expression. 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. Finally, we derive a lower bound of the worst-case complexity that shows that the upper bound from the second proof is almost tight.

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 6.3.14** Let us look at the size of the DNF or CNF for $f: \mathbb{B}^3 \rightarrow \mathbb{B}$.

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 6.3.15** Any $f: \mathbb{B}^n \rightarrow \mathbb{B}$ 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 $e_n := \text{CNF}(f)$ has $\frac{2^n}{2}$ clauses or less or $\text{DNF}(f)$ does monomials take smaller one, multiply/sum the monomials/clauses at cost $2^{n-1} - 1$ there are n literals per clause/monomial e_i , so $C(e_i) \leq 2n - 1$ so $C(e_n) \leq 2^{n-1} - 1 + 2^{n-1} \cdot (2n - 1)$ and thus $C(e_n) \in O(n \cdot 2^n)$ □



For this proof we will introduce the concept of a “realization cost function” $\kappa: \mathbb{N} \rightarrow \mathbb{N}$ 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 6.3.16** Let $\kappa(n) := \max(\{C(f) \mid f: \mathbb{B}^n \rightarrow \mathbb{B}\})$, 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: $\mathbb{B} \rightarrow \mathbb{B} = \{f_1, f_0, f_{x_1}, f_{\bar{x}_1}\}$, so $\kappa(1) = 1$ and thus $\kappa(1) \leq 2^2$.

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

P.1.2.1 given $f: \mathbb{B}^n \rightarrow \mathbb{B}$, then $f(a_1, \dots, a_n) = 1$, iff either

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

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

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

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

P.1.2.4 thus $f = f_e$, where $e := \bar{x}_n * e_0 + x_n * e_1$ and $\kappa(n) = 2 \cdot 2^n + 4 \leq 2^{n+1}$
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 6.3.17** $\kappa \in \Omega(\frac{2^n}{\log_2(n)})$

▷ **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 6.3.18** $\mathcal{E}_n := \{e \in E_{\text{bool}} \mid e \text{ has } n \text{ variables and } C(e) \leq \kappa(n)\}$

▷ **Lemma 6.3.19** $\#(\mathcal{E}_n) \geq \#(\mathbb{B}^n \rightarrow \mathbb{B})$

▷ **Proof:**

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

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

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

▷ **Corollary 6.3.20** $\#(\mathcal{E}_n) \geq 2^{2^n}$

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 $\#(\mathbb{B}^n \rightarrow \mathbb{B}) = 2^{2^n}$ \square



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

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

▷ **Lemma 6.3.22** $\sigma(n) \leq 5n$ 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 $\sigma(n) \leq \sigma(i) + \sigma(j) + 3 \leq 5 \cdot C(e_1) + 5 \cdot C(e_2) + 3 \leq 5 \cdot n - 1 + 5 = 5n$ \square

▷ **Corollary 6.3.23** $\max(\{\Lambda(e) \mid e \in \mathcal{E}_n\}) \leq 5 \cdot \kappa(n)$



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

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

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

▷ **Corollary 6.3.24** $\mathcal{E}_n \subseteq \bigcup_{i=0}^{5\kappa(n)} \mathcal{A}_n^i$ 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

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

\square



Solving for $\kappa(n)$

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

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

$$\triangleright 5\kappa(n) + 1 \cdot \log_2(n+7) \geq 2^n \quad (\text{as } \log_a(x) = \log_b(x) \cdot \log_a(b))$$

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

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

$$\triangleright \kappa(n) \in \Omega\left(\frac{2^n}{\log_2(n)}\right)$$



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 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 $f: \mathbb{B}^n \rightarrow \mathbb{B}$ (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' .



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 $f_M(c) = \text{T}$, we have, $f_{L_i}(c) = \text{T}$ for all literals in M .

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

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

▷ **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 $f \neq \lambda x.F$ 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 $f = \lambda x.T$.

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 QMC₁)
 - ▷ choose the minimal subset that covers f (sub-algorithm QMC₂)
- ▷ **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 $p := mx_i + m\bar{x}_i$ (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 $mx + m\bar{x}$ 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 QMC₁, for determining Prime Implicants

- ▷ **Definition 6.4.7** Let M be a set of monomials, then
 - ▷ $\mathcal{R}(M) := \{m \mid (mx) \in M \wedge (m\bar{x}) \in M\}$ is called the set of **resolvent** s of M
 - ▷ $\hat{\mathcal{R}}(M) := \{m \in M \mid m \text{ has a partner in } M\}$ ($n\bar{x}_i$ and nx_i are partners)
- ▷ **Definition 6.4.8 (Algorithm)** Given $f: \mathbb{B}^n \rightarrow \mathbb{B}$
 - ▷ let $M_0 := \text{DNF}(f)$ and for all $j > 0$ compute (DNF as set of monomials)
 - ▷ $M_j := \mathcal{R}(M_{j-1})$ (resolve to get sub-monomials)
 - ▷ $P_j := M_{j-1} \setminus \hat{\mathcal{R}}(M_{j-1})$ (get rid of redundant resolution partners)
 - ▷ terminate when $M_j = \emptyset$, return $P_{\text{prime}} := \bigcup_{j=1}^n P_j$



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

Example for QMC₁

x1	x2	x3	f	monomials
F	F	F	T	$x_1^0 x_2^0 x_3^0$
F	F	T	T	$x_1^0 x_2^0 x_3^1$
F	T	F	F	
F	T	T	F	
T	F	F	T	$x_1^1 x_2^0 x_3^0$
T	F	T	T	$x_1^1 x_2^0 x_3^1$
T	T	F	F	
T	T	T	T	$x_1^1 x_2^1 x_3^1$

$$P_{prime} = \bigcup_{j=1}^3 P_j = \{x_1 x_3, \overline{x_2}\}$$

$M_0 = \{ \overline{x_1 x_2 x_3}, \overline{x_1 x_2 x_3}, \overline{x_1 x_2 x_3}, \overline{x_1 x_2 x_3}, \overline{x_1 x_2 x_3} \}$
 $=: e_1^0 =: e_2^0 =: e_3^0 =: e_4^0 =: e_5^0$

$M_1 = \{ \overline{x_1 x_2}, \overline{x_2 x_3}, \overline{x_2 x_3}, \overline{x_1 x_2}, \overline{x_1 x_3} \}$
 $\mathcal{R}(e_1^0, e_2^0) \quad \mathcal{R}(e_1^0, e_3^0) \quad \mathcal{R}(e_2^0, e_4^0) \quad \mathcal{R}(e_3^0, e_4^0) \quad \mathcal{R}(e_4^0, e_5^0)$
 $=: e_1^1 =: e_2^1 =: e_3^1 =: e_4^1 =: e_5^1$

$P_1 = \emptyset$

$M_2 = \{ \overline{x_2}, \overline{x_2} \}$
 $\mathcal{R}(e_1^1, e_4^1) \quad \mathcal{R}(e_2^1, e_3^1)$

$P_2 = \{x_1 x_3\}$

$M_3 = \emptyset$

$P_3 = \{\overline{x_2}\}$

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

©: Michael Kohlhase
170

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 QMC₁

▷ **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** QMC₁ terminates after at most n rounds.

▷ **Corollary 6.4.11** P_{prime} is the set of all prime implicants of f .

©: Michael Kohlhase
171

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 algorithm s.

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 QMC₂: 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 implicants of f ,
- ▷ form a polynomial e_p as the sum over S_p and test whether $f_{e_p} = f$ and the cost of e_p is minimal

▷ **Example 6.4.13** $P_{prime} = \{x1x3, \overline{x2}\}$, so $e_p \in \{1, x1x3, \overline{x2}, x1x3 + \overline{x2}\}$.

▷ Only $f_{x1x3 + \overline{x2}} \equiv f$, so $x1x3 + \overline{x2}$ is the minimal polynomial

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


©: Michael Kohlhase
172


A better Mouse-trap for QMC₂: The Prime Implicant Table



- ▷ **Definition 6.4.14** Let $f: \mathbb{B}^n \rightarrow \mathbb{B}$ 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 $f(x) = T$
 - ▷ a central matrix of all $f_{p_i}(x)$

▷ **Example 6.4.15**

	FFF	FFT	TFF	TFT	TTT
$x1x3$	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 $f_p(c) = T$ and
 - ▷ $f_q(c) = F$ 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.


©: Michael Kohlhase
173


▷ Essential Prime Implicants and Minimal Polynomials

- ▷ **Theorem 6.4.17** Let $f: \mathbb{B}^n \rightarrow \mathbb{B}$ 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 $f(c) = T$ there is a $j \leq n$ with $f_{p_j}(c) = T$.
 - P.3** so p cannot be essential □


©: Michael Kohlhase
174


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	$x_1^0 x_2^0 x_3^0 x_4^0$
F	F	F	T	T	$x_1^0 x_2^0 x_3^0 x_4^1$
F	F	T	F	T	$x_1^0 x_2^0 x_3^1 x_4^0$
F	F	T	T	F	
F	T	F	F	F	
F	T	F	T	T	$x_1^0 x_2^1 x_3^0 x_4^1$
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	$x_1^1 x_2^0 x_3^1 x_4^0$
T	F	T	T	T	$x_1^1 x_2^0 x_3^1 x_4^1$
T	T	F	F	F	
T	T	F	T	F	
T	T	T	F	T	$x_1^1 x_2^1 x_3^1 x_4^0$
T	T	T	T	T	$x_1^1 x_2^1 x_3^1 x_4^1$



©: Michael Kohlhase

175



A complex Example for QMC (QMC_1)

$$M_0 = \{x_1^0 x_2^0 x_3^0 x_4^0, x_1^0 x_2^0 x_3^0 x_4^1, x_1^0 x_2^0 x_3^1 x_4^0, x_1^0 x_2^1 x_3^0 x_4^1, x_1^1 x_2^0 x_3^1 x_4^0, x_1^1 x_2^0 x_3^1 x_4^1, x_1^1 x_2^1 x_3^1 x_4^0, x_1^1 x_2^1 x_3^1 x_4^1\}$$

$$M_1 = \{x_1^0 x_2^0 x_3^0, x_1^0 x_2^0 x_4^0, x_1^0 x_3^0 x_4^1, x_1^1 x_2^0 x_3^1, x_1^1 x_2^1 x_3^1, x_1^1 x_3^1 x_4^1, x_2^0 x_3^1 x_4^0, x_1^1 x_3^1 x_4^0\}$$

$$P_1 = \emptyset$$

$$M_2 = \{x_1^1 x_3^1\}$$

$$P_2 = \{x_1^0 x_2^0 x_3^0, x_1^0 x_2^0 x_4^0, x_1^0 x_3^0 x_4^1, x_2^0 x_3^1 x_4^0\}$$

$$M_3 = \emptyset$$

$$P_3 = \{x_1^1 x_3^1\}$$

$$P_{\text{prime}} = \{\overline{x_1} \overline{x_2} \overline{x_3}, \overline{x_1} \overline{x_2} \overline{x_4}, \overline{x_1} \overline{x_3} x_4, \overline{x_2} x_3 \overline{x_4}, x_1 x_3\}$$



©: Michael Kohlhase

176




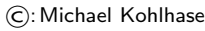
A better Mouse-trap for QMC_1 : optimizing the data structure


▷ Idea: Do the calculations directly on the DNF table


x1	x2	x3	x4	monomials
F	F	F	F	$x_1^0 x_2^0 x_3^0 x_4^0$
F	F	F	T	$x_1^0 x_2^0 x_3^0 x_4^1$
F	F	T	F	$x_1^0 x_2^0 x_3^1 x_4^0$
F	T	F	T	$x_1^0 x_2^1 x_3^0 x_4^1$
T	F	T	F	$x_1^1 x_2^0 x_3^1 x_4^0$
T	F	T	T	$x_1^1 x_2^0 x_3^1 x_4^1$
T	T	T	F	$x_1^1 x_2^1 x_3^1 x_4^0$
T	T	T	T	$x_1^1 x_2^1 x_3^1 x_4^1$

- ▷ **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** $\langle F, F, F, F \rangle$ and $\langle F, F, F, T \rangle$ resolve to $\langle F, F, F, X \rangle$.









A better Mouse-trap for QMC₁: optimizing the data structure

- ▷ One step resolution on the table

x1	x2	x3	x4	monomials
F	F	F	F	$x_1^0 x_2^0 x_3^0 x_4^0$
F	F	F	T	$x_1^0 x_2^0 x_3^0 x_4^1$
F	F	T	F	$x_1^0 x_2^0 x_3^1 x_4^0$
F	T	F	T	$x_1^0 x_2^1 x_3^0 x_4^1$
T	F	T	F	$x_1^1 x_2^0 x_3^1 x_4^0$
T	F	T	T	$x_1^1 x_2^0 x_3^1 x_4^1$
T	T	T	F	$x_1^1 x_2^1 x_3^1 x_4^0$
T	T	T	T	$x_1^1 x_2^1 x_3^1 x_4^1$


~

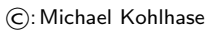
x1	x2	x3	x4	monomials
F	F	F	X	$x_1^0 x_2^0 x_3^0$
F	F	X	F	$x_1^0 x_2^0 x_4^0$
F	X	F	T	$x_1^0 x_3^0 x_4^1$
T	F	T	X	$x_1^1 x_2^0 x_3^1$
T	T	T	X	$x_1^1 x_2^1 x_3^1$
T	X	T	T	$x_1^1 x_3^1 x_4^1$
X	F	T	F	$x_2^0 x_3^1 x_4^0$
T	X	T	F	$x_1^1 x_3^1 x_4^0$


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


x1	x2	x3	x4	monomials
F	F	F	X	$x_1^0 x_2^0 x_3^0$
F	F	X	F	$x_1^0 x_2^0 x_4^0$
F	X	F	T	$x_1^0 x_3^0 x_4^1$
T	X	T	X	$x_1^1 x_3^1$
X	F	T	F	$x_2^0 x_3^1 x_4^0$

- ▷ This table represents the prime implicants of f










A complex Example for QMC (QMC₁)


- ▷ The PIT:


	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x_1} x_2 x_3$	T	T	F	F	F	F	F	F
$\overline{x_1} x_2 x_4$	T	F	T	F	F	F	F	F
$\overline{x_1} x_3 x_4$	F	T	F	T	F	F	F	F
$\overline{x_2} x_3 x_4$	F	F	T	F	T	F	F	F
$x_1 x_3$	F	F	F	F	T	T	T	T
- ▷ $\overline{x_1} \overline{x_2} \overline{x_3}$ is not essential, so we are left with


	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x_1} x_2 x_4$	T	F	T	F	F	F	F	F
$\overline{x_1} x_3 x_4$	F	T	F	T	F	F	F	F
$\overline{x_2} x_3 x_4$	F	F	T	F	T	F	F	F
$x_1 x_3$	F	F	F	F	T	T	T	T
- ▷ here $\overline{x_2}, x_3, \overline{x_4}$ is not essential, so we are left with

	FFFF	FFFT	FFTF	FTFT	TFTF	TFTT	TTTF	TTTT
$\overline{x_1} x_2 x_4$	T	F	T	F	F	F	F	F
$\overline{x_1} x_3 x_4$	F	T	F	T	F	F	F	F
$x_1 x_3$	F	F	F	F	T	T	T	T
- ▷ all the remaining ones ($\overline{x_1} \overline{x_2} \overline{x_4}$, $\overline{x_1} \overline{x_3} x_4$, and $x_1 x_3$) are essential









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



©: Michael Kohlhase

179



⚠ 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		3 vars				4 vars			
	\overline{A} A		\overline{AB} \overline{AB} AB AB	\overline{CD}	\overline{AB}	\overline{AB}	AB	AB	
\overline{B}		\overline{C}		\overline{CD}	m_0	m_4	m_{12}	m_8	
B		C		CD	m_1	m_5	m_{13}	m_9	
				CD	m_3	m_7	m_{15}	m_{11}	
				\overline{CD}	m_2	m_6	m_{14}	m_{10}	
	square		ring		torus				
	2/4-groups		2/4/8-groups		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)



©: Michael Kohlhase

180



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

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



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 (proposition s) 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

$$[\neg: \mathbb{B} \rightarrow \mathbb{B}], [\vee: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}] \in \mathcal{D}$$

then we call the set $\mathcal{T}_{\mathbb{B}}^g(\Sigma)$ 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 $\mathcal{T}_{\mathbb{B}}^g(\Sigma)$ without $\vee, \wedge, \neg, \Rightarrow,$ and \Leftrightarrow **atom s**. (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 $\neg \mathbf{A} \vee \mathbf{B} \dots$
- ▷ Build print routine $\hat{\cdot}$ with $\widehat{\mathbf{A} \wedge \mathbf{B}} = \widehat{\mathbf{A}} * \widehat{\mathbf{B}}$, and $\widehat{\neg \mathbf{A}} = \overline{\widehat{\mathbf{A}}}$ and that turns atoms into variable names. (variables and atoms are countable)



Conventions for Brackets in Propositional Logic

- ▷ **we leave out outer brackets:** $\mathbf{A} \Rightarrow \mathbf{B}$ abbreviates $(\mathbf{A} \Rightarrow \mathbf{B})$.
- ▷ **implications are right associative:** $\mathbf{A}^1 \Rightarrow \dots \Rightarrow \mathbf{A}^n \Rightarrow \mathbf{C}$ abbreviates $\mathbf{A}^1 \Rightarrow \dots \Rightarrow \dots \Rightarrow \mathbf{A}^n \Rightarrow \mathbf{C}$
- ▷ 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 reasoning¹.

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 $\mathcal{I}_\varphi(e) = \mathbf{T}$ (write $\mathcal{M} \models^\varphi e$)
 - ▷ **false under φ** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \mathbf{F}$ (write $\mathcal{M} \not\models^\varphi e$)
 - ▷ **satisfiable** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \mathbf{T}$ 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 $\mathcal{I}_\varphi(e) = \mathbf{F}$ for some assignments φ
 - ▷ **unsatisfiable** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \mathbf{F}$ 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 clear)
- ▷ **Definition 7.1.7 (Entailment)** (aka. logical consequence)
 We say that e **entails** f ($e \models f$), iff $\mathcal{I}_\varphi(f) = \mathbf{T}$ for all φ with $\mathcal{I}_\varphi(e) = \mathbf{T}$ (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

¹Here (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.

examples counterexamples, and such assertions “falsifiable”. We also often give examples for certain assertions to show that they can indeed be made true (which is not the same as being valid yet); such assertions we call “satisfiable”. Finally, if an assertion cannot be made true in any circumstances we call it “unsatisfiable”; such assertions naturally arise in mathematical practice in the form of refutation proofs, where we show that an assertion (usually the negation of the theorem we want to prove) leads to an obviously unsatisfiable conclusion, showing that the negation of the theorem is unsatisfiable, and thus the theorem valid.

Example: Propositional Logic with ADT variables

- ▷ **Idea:** We use propositional logic to express things about the world ($\text{PLNQ} \hat{=} \text{Predicate Logic without Quantifiers}$)
- ▷ **Example 7.1.8** Abstract Data Type: $\langle \{\mathbb{B}, \mathbb{I}\}, \{ \dots, [\text{love}: \mathbb{I} \times \mathbb{I} \rightarrow \mathbb{B}], [\text{bill}: \mathbb{I}], [\text{mary}: \mathbb{I}], \dots \rangle$
ground terms:
 - ▷ $g_1 := \text{love}(\text{bill}, \text{mary})$ (how nice)
 - ▷ $g_2 := \text{love}(\text{mary}, \text{bill}) \wedge \neg \text{love}(\text{bill}, \text{mary})$ (how sad)
 - ▷ $g_3 := \text{love}(\text{bill}, \text{mary}) \wedge \text{love}(\text{mary}, \text{john}) \Rightarrow \text{hate}(\text{bill}, \text{john})$ (how natural)
- ▷ **Semantics:** by mapping into known stuff, (e.g. \mathbb{I} to persons \mathbb{B} to $\{\text{T}, \text{F}\}$)
- ▷ **Idea:** Import semantics from Boolean Algebra (atoms “are” variables)
 - ▷ only need variable assignment $\varphi: \mathcal{A}(\Sigma) \rightarrow \{\text{T}, \text{F}\}$
- ▷ **Example 7.1.9** $\mathcal{I}_\varphi(\text{love}(\text{bill}, \text{mary}) \wedge (\text{love}(\text{mary}, \text{john}) \Rightarrow \text{hate}(\text{bill}, \text{john}))) = \text{T}$ if $\varphi(\text{love}(\text{bill}, \text{mary})) = \text{T}$, $\varphi(\text{love}(\text{mary}, \text{john})) = \text{F}$, and $\varphi(\text{hate}(\text{bill}, \text{john})) = \text{T}$
- ▷ **Example 7.1.10** $g_1 \wedge g_3 \wedge \text{love}(\text{mary}, \text{john}) \models \text{hate}(\text{bill}, \text{john})$



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 ($\llbracket \text{three plus five} \rrbracket = 8$)
 - ▷ **Validity:** $\mathcal{M} \models \mathbf{A}$, iff $\llbracket \mathbf{A} \rrbracket^{\mathcal{M}} = \text{T}$ (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?



©: Michael Kohlhase

187



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, \dots and **implication:** \Rightarrow
- ▷ **Semantics:** $\mathcal{I}_\varphi(P) = \varphi(P)$ and $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \top$, iff $\mathcal{I}_\varphi(\mathbf{A}) = \text{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \top$.
- ▷ **K** := $P \Rightarrow Q \Rightarrow P$, **S** := $(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R$
- ▷
$$\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}} \text{MP} \qquad \frac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})} \text{Subst}$$
- ▷ Let us look at a \mathcal{H}^0 theorem (with a proof)
- ▷ $\mathbf{C} \Rightarrow \mathbf{C}$ (*Tertium non datur*)

▷ **Proof:**

P.1 $(C \Rightarrow (C \Rightarrow C) \Rightarrow C) \Rightarrow (C \Rightarrow C \Rightarrow C) \Rightarrow C \Rightarrow C$ (S with $[C/P], [C \Rightarrow C/Q], [C/R]$)



P.2 $C \Rightarrow (C \Rightarrow C) \Rightarrow C$ (K with $[C/P], [C \Rightarrow C/Q]$)

P.3 $(C \Rightarrow C \Rightarrow C) \Rightarrow C \Rightarrow C$ (MP on P.1 and P.2)

P.4 $C \Rightarrow C \Rightarrow C$ (K with $[C/P], [C/Q]$)

P.5 $C \Rightarrow C$ (MP on P.3 and P.4)

P.6 We have shown that $\emptyset \vdash_{\mathcal{H}^0} C \Rightarrow C$ (i.e. $C \Rightarrow C$ is a **theorem**) (is is also valid?) □


©: Michael Kohlhase
188


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

$$(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R \quad (7.1)$$

which we can always do, since we have an axiom for this formula, then we apply the rule *subst*, where **A** is this result, **B** is C , and X is the variable P to obtain

$$(C \Rightarrow Q \Rightarrow R) \Rightarrow (C \Rightarrow Q) \Rightarrow C \Rightarrow R \quad (7.2)$$

Next we apply the rule *subst* to this where **B** is $C \Rightarrow C$ and X is the variable Q this time to obtain

$$(C \Rightarrow (C \Rightarrow C) \Rightarrow R) \Rightarrow (C \Rightarrow C \Rightarrow C) \Rightarrow C \Rightarrow R \quad (7.3)$$

And again, we apply the rule *subst* this time, **B** is 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 $'a \rightarrow 'b \rightarrow 'a$ 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 Funktion”, so you have letter K in the name. For the **S** axiom (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}

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

where $\mathbf{A}_1, \dots, \mathbf{A}_n$ and \mathbf{C} are formula schemata for \mathcal{L} and \mathcal{N} is a name. The \mathbf{A}_i are called **assumption** s, 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} .

 ©:Michael Kohlhase 189 

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 $\mathbf{A}_1, \dots, \mathbf{A}_m$ 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 $\frac{\mathbf{A}_{l_1} \cdots \mathbf{A}_{l_k}}{\mathbf{A}_i}$ 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.



$$\frac{\frac{P \Rightarrow Q \Rightarrow P \quad K}{P \Rightarrow Q \Rightarrow P} \quad P}{Q \Rightarrow P} MP$$

▷ **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 $\vdash_{\mathcal{C}}$ defined in Definition 7.3.6 is a **derivation relation** in the sense of Definition 7.3.1.²

▷ **Definition 7.3.9** 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.

 ©:Michael Kohlhase 190 

^bEDNOTE: MK: this should become a view!

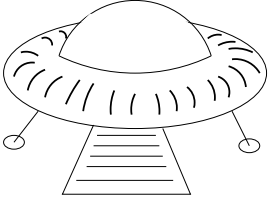
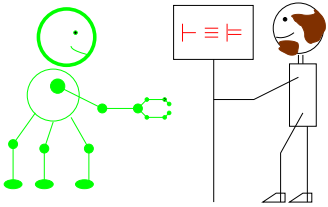
Inference rules are relations on formulae represented by formula schemata (where boldface, uppercase letters are used as meta-variables for formulae). For instance, in Example 7.3.7 the inference rule $\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}$ was applied in a situation,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



©: Michael Kohlhase

191



JACOBS UNIVERSITY

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.

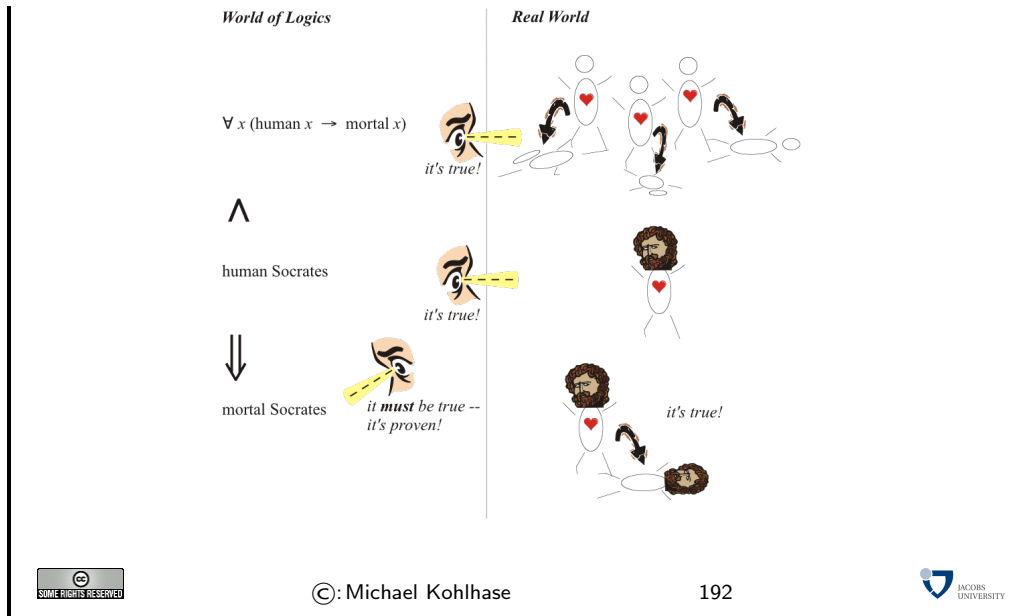
Of course, the logics we have studied so far are very simple, and not able to express interesting facts about the world, but we will study them as a simple example of the fundamental problem of Computer Science: How do the formal representations correlate with the real world.

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 *Socrates 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** $\vdash A$ implies $\models A$ for all propositions 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 $A \Rightarrow B$ be valid, and $\varphi: \mathcal{V}_o \rightarrow \{T, F\}$ arbitrary
 - P.2.2.2 then $\mathcal{I}_\varphi(A) = F$ or $\mathcal{I}_\varphi(B) = T$ (by definition of \Rightarrow).
 - P.2.2.3 Since A is valid, $\mathcal{I}_\varphi(A) = T \neq F$, so $\mathcal{I}_\varphi(B) = T$.
 - P.2.2.4 As φ was arbitrary, 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	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	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

□

©: Michael Kohlhase
194

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 $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathbf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{C})) = \mathbf{F}$ or $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{D})) = \mathbf{T}$.

P.1.2.2 This is the case, iff $\mathcal{I}_\psi(\mathbf{C}) = \mathbf{F}$ or $\mathcal{I}_\psi(\mathbf{D}) = \mathbf{T}$ by IH (\mathbf{C} and \mathbf{D} have smaller depth than \mathbf{A}).

P.1.2.3 In other words, $\mathcal{I}_\psi(\mathbf{A}) = \mathcal{I}_\psi(\mathbf{C} \Rightarrow \mathbf{D}) = \mathbf{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathbf{T}$ 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 preserves 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 $[B/X](A)$ is valid, if A is.

P.1 Let A be valid, B a formula, $\varphi: \mathcal{V}_o \rightarrow \{T, F\}$ a variable assignment, and $\psi := \varphi, [I_\varphi(B)/X]$.

P.2 then $I_\varphi([B/X](A)) = I_{\varphi, [I_\varphi(B)/X]}(A) = T$, since A is valid.

P.3 As the argumentation did not depend on the choice of φ , $[B/X](A)$ valid and we have proven the assertion. \square



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 $\mathcal{H}, A \models B$, then $\mathcal{H} \models (A \Rightarrow B)$.*

▷ **Proof:** We show that $I_\varphi(A \Rightarrow B) = T$ for all assignments φ with $I_\varphi(\mathcal{H}) = T$ whenever $\mathcal{H}, A \models B$

P.1 Let us assume there is an assignment φ , such that $I_\varphi(A \Rightarrow B) = F$.

P.2 Then $I_\varphi(A) = T$ and $I_\varphi(B) = F$ by definition.

P.3 But we also know that $I_\varphi(\mathcal{H}) = T$ and thus $I_\varphi(B) = T$, since $\mathcal{H}, A \models B$.

P.4 This contradicts our assumption $I_\varphi(B) = F$ from above.

P.5 So there cannot be an assignment φ that $I_\varphi(A \Rightarrow B) = F$; in other words, $A \Rightarrow B$ is valid. \square



Now, we complete the theorem by proving the converse direction, which is rather simple.

The Entailment Theorem (continued)

▷ **Corollary 7.4.6** *$\mathcal{H}, A \models B$, iff $\mathcal{H} \models (A \Rightarrow B)$*

▷ **Proof:** In the light of the previous result, we only need to prove that $\mathcal{H}, A \models B$,

whenever $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$

P.1 To prove that $\mathcal{H}, \mathbf{A} \models \mathbf{B}$ we assume that $\mathcal{I}_\varphi(\mathcal{H}, \mathbf{A}) = \text{T}$.

P.2 In particular, $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \text{T}$ since $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.

P.3 Thus we have $\mathcal{I}_\varphi(\mathbf{A}) = \text{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \text{T}$.

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 \mathbf{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 $\mathbf{C}_1, \dots, \mathbf{C}_m$ be a proof of \mathbf{B} from the hypotheses \mathcal{H} .

P.2 We generalize the induction hypothesis: For all $1 \leq i \leq m$ 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 $[B/X]$ would have to be applied to 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:** C_i is inferred by Subst from C_j for $j < i$ with $[B/X]$.

P.1 So $C_i = [B/X](C_j)$; we have $\mathcal{H} \vdash A \Rightarrow C_j$ by IH

P.2 so by Subst we have $\mathcal{H} \vdash [B/X](A \Rightarrow C_j)$. (Oooops! $\neq A \Rightarrow C_i$) □


©: Michael Kohlhase
200




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)

$A \Rightarrow B \Rightarrow A, (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
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 $\vdash_{\mathcal{H}^1}$


©: Michael Kohlhase
201


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 $\mathcal{H}, A \vdash_{\mathcal{H}^1} B$, then $\mathcal{H} \vdash_{\mathcal{H}^1} A \Rightarrow B$
- ▷ **Proof:** Let C_1, \dots, C_m be a proof of B from the hypotheses \mathcal{H} .

P.1 We construct proofs $\mathcal{H} \vdash_{\mathcal{H}^1} A \Rightarrow C_i$ for all $1 \leq i \leq n$ by induction on i .

P.2 We have to consider three cases

P.2.1 C_i is an axiom or hypothesis:

P.2.1.1 Then $\mathcal{H} \vdash_{\mathcal{H}^1} C_i$ by construction and $\mathcal{H} \vdash_{\mathcal{H}^1} C_i \Rightarrow A \Rightarrow C_i$ by Ax1.

P.2.1.2 So $\mathcal{H} \vdash_{\mathcal{H}^1} C_i$ by MP □

P.2.2 $C_i = A$:

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 $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ \square

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) \square

\square

\square



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 calculi 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 $\dots \mathbf{A} \Rightarrow \mathbf{B} \dots \mathbf{A} \dots \mathbf{B} \dots [\mathbf{C}/\mathbf{X}](\mathbf{B}) \dots$ is replaced by

$\dots \mathbf{A} \Rightarrow \mathbf{B} \dots [\mathbf{C}/\mathbf{X}](\mathbf{A}) \Rightarrow [\mathbf{C}/\mathbf{X}](\mathbf{B}) \dots \mathbf{A} \dots [\mathbf{C}/\mathbf{X}](\mathbf{A}) \dots [\mathbf{C}/\mathbf{X}](\mathbf{B}) \dots$

P.5 Thus we can push later Subst applications to the axioms, transforming a \mathcal{H}^0 proof into a \mathcal{H}^1 proof. \square

▷ **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 $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$ \square



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 A, B , we have $A \vdash_{\mathcal{H}^0} B$ implies $A \models B$.

▷ **Proof:**

P.1 By deduction theorem $A \vdash_{\mathcal{H}^0} B$, iff $\vdash A \Rightarrow B$,

P.2 by the first soundness theorem this is the case, iff $\models A \Rightarrow B$,

P.3 by the entailment theorem this holds, iff $A \models B$. □

 ©: Michael Kohlhase 204 

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

Properties of Calculi (Theoretical Logic)

▷ **Correctness:** (provable implies valid)

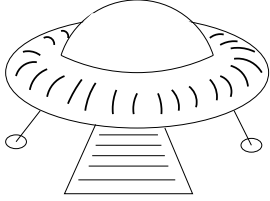
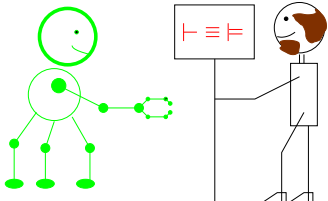
▷ $\mathcal{H} \vdash B$ implies $\mathcal{H} \models B$ (equivalent: $\vdash A$ implies $\models A$)



▷ **Completeness:** (valid implies provable)

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

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

▷ **To TRUTH through PROOF** (CALCULEMUS [Leibniz ~1680])

 ©: Michael Kohlhase 205 

7.5 A Calculus for Mathtalk

In our introduction to Section 7.0 we have positioned Boolean expressions (and proposition logic) as a system for understanding the mathematical language “mathtalk” introduced in Section 3.3. We have been using this language to state properties of objects and prove them all through this course without making the rules that govern this activity fully explicit. We will rectify this now: First we give a calculus that tries to mimic the informal rules mathematicians use in 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

<p>Introduction</p> $\frac{\mathbf{A} \quad \mathbf{B}}{\mathbf{A} \wedge \mathbf{B}} \wedge I$	<p>Elimination</p> $\frac{\mathbf{A} \wedge \mathbf{B}}{\mathbf{A}} \wedge E_l \quad \frac{\mathbf{A} \wedge \mathbf{B}}{\mathbf{B}} \wedge E_r$	<p>Axiom</p> $\frac{}{\mathbf{A} \vee \neg \mathbf{A}} \text{TND}$
$\frac{\begin{array}{c} \underline{\underline{[\mathbf{A}]^1}} \\ \mathbf{B} \end{array}}{\mathbf{A} \Rightarrow \mathbf{B}} \Rightarrow I^1$	$\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}} \Rightarrow E$	

▷ TND is used only in classical logic (otherwise constructive/intuitionistic)



The most characteristic rule in the natural deduction calculus is the $\Rightarrow I$ rule. It corresponds to the mathematical way of proving an implication $\mathbf{A} \Rightarrow \mathbf{B}$: We assume that \mathbf{A} is true and show \mathbf{B} from this assumption. When we can do this we discharge (get rid of) the assumption and conclude $\mathbf{A} \Rightarrow \mathbf{B}$. This mode of reasoning is called **hypothetical reasoning**. Note that the local hypothesis is **discharged** by the rule $\Rightarrow I$, i.e. it cannot be used in any other part of the proof. As the $\Rightarrow I$ rules may be nested, we decorate both the rule and the corresponding assumption with a marker (here the number 1).

Let us now consider an example of hypothetical reasoning in action.

Natural Deduction: Examples

▷ Inference with local hypotheses

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

$$\frac{\frac{[\mathbf{A}]^1}{\mathbf{B} \Rightarrow \mathbf{A}} \Rightarrow I^2}{\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}} \Rightarrow I^1$$



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 $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$ by weakening and $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$ 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.

$$\frac{\mathbf{A}}{\mathbf{A} \vee \mathbf{B}} \vee I_l \quad \frac{\mathbf{B}}{\mathbf{A} \vee \mathbf{B}} \vee I_r \quad \frac{\begin{array}{c} [\mathbf{A}]^1 \quad [\mathbf{B}]^1 \\ \vdots \quad \vdots \\ \mathbf{A} \vee \mathbf{B} \quad \mathbf{C} \end{array}}{\mathbf{C}} \vee E^1$$

$$\frac{\begin{array}{c} [\mathbf{A}]^1 \\ \vdots \\ \mathbf{F} \end{array}}{\neg \mathbf{A}} \neg I^1 \quad \frac{\neg \neg \mathbf{A}}{\mathbf{A}} \neg E$$

$$\frac{\neg \mathbf{A} \quad \mathbf{A}}{\mathbf{F}} FI \quad \frac{\mathbf{F}}{\mathbf{A}} FE$$

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 is 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.

³EDNOTE: 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

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

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

* 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.

Natural Deduction with Equality

- ▷ **Definition 7.5.5 (First-Order Logic with Equality)** We extend PL^1 with a new logical symbol for equality $= \in \Sigma_2^p$ and fix its semantics to $\mathcal{I}(=) := \{\langle x, x \rangle \mid x \in \mathcal{D}_i\}$. We call the extended logic **first-order logic with equality** ($\text{PL}_{=}^1$)
- ▷ We now extend natural deduction as well.
- ▷ **Definition 7.5.6** For the calculus of natural deduction with equality $\mathcal{ND}_{=}^1$ 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} .



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

With the \mathcal{MD}^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{MD}^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 7.5.7 $\sqrt{2}$ 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{MD}^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{MD}^1 inference rules. Figure 7.1 shows such a proof, where we write n is prime, use $\#(n)$ for the number of prime factors of a number n , and write $\text{irr}(r)$ 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{MD}^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{MD}^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 Section 7.2 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		$\forall n, p. p \Rightarrow \#(pn) = \#(n) + 1$	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	$r^2 \Rightarrow \#(2s^2) = \#(s^2) + 1$	$\forall E^2(1)$
13		r^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{MD}^1 proof of the irrationality of $\sqrt{2}$

Chapter 8

Machine-Oriented Calculi

Now we have studied the Hilbert-style calculus in some detail, let us look at two calculi that work via a totally different principle. Instead of deducing new formulae from axioms (and hypotheses) and hoping to arrive at the desired theorem, we try to deduce a contradiction from the negation of the theorem. Indeed, a formula \mathbf{A} is valid, iff $\neg\mathbf{A}$ is unsatisfiable, so if we derive a contradiction from $\neg\mathbf{A}$, then we have proven \mathbf{A} . The advantage of such “test-calculi” (also called negative calculi) is easy to see. Instead of finding a proof that ends in \mathbf{A} , we have to find any of a broad class of contradictions. This makes the calculi that we will discuss now easier to control and therefore more suited for mechanization.

8.1 Calculi for Automated Theorem Proving: Analytical Tableaux

8.1.1 Analytical Tableaux

Before we can start, we will need to recap some nomenclature on formulae.

Recap: Atoms and Literals

- ▷ **Definition 8.1.1** We call a formula **atomic**, or an **atom**, iff it does not contain connectives. We call a formula **complex**, iff it is not atomic.
- ▷ **Definition 8.1.2** We call a pair \mathbf{A}^α a **labeled formula**, if $\alpha \in \{\mathbf{T}, \mathbf{F}\}$. A labeled atom is called **literal**.
- ▷ **Definition 8.1.3** Let Φ be a set of formulae, then we use $\Phi^\alpha := \{\mathbf{A}^\alpha \mid \mathbf{A} \in \Phi\}$.



©: Michael Kohlhase

212



The idea about literals is that they are atoms (the simplest formulae) that carry around their intended truth value.

Now we will also review some propositional identities that will be useful later on. Some of them we have already seen, and some are new. All of them can be proven by simple truth table arguments.

Test Calculi: Tableaux and Model Generation

▷ **Idea:** instead of showing $\emptyset \vdash Th$, show $\neg Th \vdash trouble$ (use \perp for trouble)

▷ **Example 8.1.4** Tableau Calculi try to construct models.

Tableau Refutation (Validity)	Model generation (Satisfiability)
$\models P \wedge Q \Rightarrow Q \wedge P$	$\models P \wedge (Q \vee \neg R) \wedge \neg Q$
$ \begin{array}{c} P \wedge Q \Rightarrow Q \wedge P^f \\ P \wedge Q^t \\ Q \wedge P^f \\ P^t \\ Q^t \\ P^f \mid Q^f \\ \perp \mid \perp \end{array} $	$ \begin{array}{c} P \wedge (Q \vee \neg R) \wedge \neg Q^t \\ P \wedge (Q \vee \neg R)^t \\ \neg Q^t \\ Q^f \\ P^t \\ Q \vee \neg R^t \\ Q^t \mid \neg R^t \\ \perp \mid R^f \end{array} $
No Model	Herbrand Model $\{P^t, Q^f, R^f\}$ $\varphi := \{P \mapsto T, Q \mapsto F, R \mapsto F\}$

Algorithm: Fully expand all possible tableaux, (no rule can be applied)
 ▷ **Satisfiable**, iff there are open branches (correspond to models)



Tableau calculi develop a formula in a tree-shaped arrangement that represents a case analysis on when a formula can be made true (or false). Therefore the formulae are decorated with exponents that hold the intended truth value.

On the left we have a refutation tableau that analyzes a negated formula (it is decorated with the intended truth value F). Both branches contain an elementary contradiction \perp .

On the right we have a model generation tableau, which analyzes a positive formula (it is decorated with the intended truth value T. This tableau uses the same rules as the refutation tableau, but makes a case analysis of when this formula can be satisfied. In this case we have a closed branch and an open one, which corresponds a model).

Now that we have seen the examples, we can write down the tableau rules formally.



Analytical Tableaux (Formal Treatment of \mathcal{T}_0)

- ▷ formula is analyzed in a tree to determine satisfiability
- ▷ branches correspond to valuations (models)
- ▷ one per connective

$$\frac{\mathbf{A} \wedge \mathbf{B}^t}{\mathbf{A}^t \mid \mathbf{B}^t} \mathcal{T}_0 \wedge \quad \frac{\mathbf{A} \wedge \mathbf{B}^f}{\mathbf{A}^f \mid \mathbf{B}^f} \mathcal{T}_0 \vee \quad \frac{\neg \mathbf{A}^t}{\mathbf{A}^f} \mathcal{T}_0 \neg^T \quad \frac{\neg \mathbf{A}^f}{\mathbf{A}^t} \mathcal{T}_0 \neg^F \quad \frac{\mathbf{A}^\alpha}{\mathbf{A}^\beta} \alpha \neq \beta}{\perp} \mathcal{T}_0 \text{cut}$$

- ▷ Use rules exhaustively as long as they contribute new material
- ▷ **Definition 8.1.5** Call a tableau **saturated**, iff no rule applies, and a branch **closed**, iff it ends in \perp , else **open**. (open branches in saturated tableaux yield models)
- ▷ **Definition 8.1.6 (\mathcal{T}_0 -Theorem/Derivability)** \mathbf{A} is a \mathcal{T}_0 -theorem ($\vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau with \mathbf{A}^F at the root.

$\Phi \subseteq \text{wff}_o(\mathcal{V}_o)$ derives \mathbf{A} in \mathcal{T}_0 ($\Phi \vdash_{\mathcal{T}_0} \mathbf{A}$), iff there is a closed tableau starting with \mathbf{A}^F and Φ^T .


©: Michael Kohlhase
214


These inference rules act on tableaux have to be read as follows: if the formulae over the line appear in a tableau branch, then the branch can be extended by the formulae or branches below the line. There are two rules for each primary connective, and a branch closing rule that adds the special symbol \perp (for unsatisfiability) to a branch.

We use the tableau rules with the convention that they are only applied, if they contribute new material to the branch. This ensures termination of the tableau procedure for propositional logic (every rule eliminates one primary connective).

Definition 8.1.7 We will call a closed tableau with the signed formula \mathbf{A}^α at the root a **tableau refutation** for \mathcal{A}^α .

The saturated tableau represents a full case analysis of what is necessary to give \mathbf{A} the truth value α ; since all branches are closed (contain contradictions) this is impossible.

Definition 8.1.8 We will call a tableau refutation for \mathbf{A}^f a **tableau proof** for \mathbf{A} , since it refutes the possibility of finding a model where \mathbf{A} evaluates to F. Thus \mathbf{A} must evaluate to T in all models, which is just our definition of validity.

Thus the tableau procedure can be used as a calculus for propositional logic. In contrast to the calculus in section ?sec.hilbert? it does not prove a theorem \mathbf{A} by deriving it from a set of axioms, but it proves it by refuting its negation. Such calculi are called negative or test calculi. Generally negative calculi have computational advantages over positive ones, since they have a built-in sense of direction.

We have rules for all the necessary connectives (we restrict ourselves to \wedge and \neg , since the others can be expressed in terms of these two via the propositional identities above. For instance, we can write $\mathbf{A} \vee \mathbf{B}$ as $\neg(\neg\mathbf{A} \wedge \neg\mathbf{B})$, and $\mathbf{A} \Rightarrow \mathbf{B}$ as $\neg\mathbf{A} \vee \mathbf{B}, \dots$)

We will now look at an example. Following our introduction of propositional logic in in Example 7.1.9 we look at a formulation of propositional logic with fancy variable names. Note that love(mary, bill) is just a variable name like P or X , which we have used earlier.

A Valid Real-World Example

▷ **Example 8.1.9** *If Mary loves Bill and John loves Mary, then John loves Mary*

$$\begin{array}{l}
 \text{love(mary, bill)} \wedge \text{love(john, mary)} \Rightarrow \text{love(john, mary)}^f \\
 \neg(\neg\neg(\text{love(mary, bill)} \wedge \text{love(john, mary)}) \wedge \neg\text{love(john, mary)})^f \\
 \neg\neg(\text{love(mary, bill)} \wedge \text{love(john, mary)}) \wedge \neg\text{love(john, mary)}^t \\
 \neg\neg(\text{love(mary, bill)} \wedge \text{love(john, mary)})^t \\
 \neg(\text{love(mary, bill)} \wedge \text{love(john, mary)})^f \\
 \text{love(mary, bill)} \wedge \text{love(john, mary)}^t \\
 \neg\text{love(john, mary)}^t \\
 \text{love(mary, bill)}^t \\
 \text{love(john, mary)}^t \\
 \text{love(john, mary)}^f \\
 \perp
 \end{array}$$

This is a closed tableau, so the $\text{love(mary, bill)} \wedge \text{love(john, mary)} \Rightarrow \text{love(john, mary)}$ is a \mathcal{T}_0 -theorem.



Again, the derivability version is much simpler

Testing for Entailment in \mathcal{T}_0

▷ **Example 8.1.12** Does *Mary loves Bill or John loves Mary* entail that *John loves Mary*?

$$\begin{array}{c|c} \text{love(mary, bill)} \vee \text{love(john, mary)}^t & \\ \text{love(john, mary)}^f & \\ \text{love(mary, bill)}^t & \text{love(john, mary)}^t \\ \hline & \perp \end{array}$$

This saturated tableau has an open branch that shows that the interpretation with $\mathcal{I}_\varphi(\text{love(mary, bill)}) = T$ but $\mathcal{I}_\varphi(\text{love(john, mary)}) = F$ falsifies the derivability/entailment conjecture.


©: Michael Kohlhase
218




8.1.2 Practical Enhancements for Tableaux

Propositional Identities

▷ **Definition 8.1.13** Let T and F be new logical constants with $\mathcal{I}(T) = T$ and $\mathcal{I}(F) = F$ for all assignments φ .

▷ We have to following identities:

Name	for \wedge	for \vee
Idenpotence	$\varphi \wedge \varphi = \varphi$	$\varphi \vee \varphi = \varphi$
Identity	$\varphi \wedge T = \varphi$	$\varphi \vee F = \varphi$
Absorption I	$\varphi \wedge F = F$	$\varphi \vee T = T$
Commutativity	$\varphi \wedge \psi = \psi \wedge \varphi$	$\varphi \vee \psi = \psi \vee \varphi$
Associativity	$\varphi \wedge (\psi \wedge \theta) = (\varphi \wedge \psi) \wedge \theta$	$\varphi \vee (\psi \vee \theta) = (\varphi \vee \psi) \vee \theta$
Distributivity	$\varphi \wedge (\psi \vee \theta) = \varphi \wedge \psi \vee \varphi \wedge \theta$	$\varphi \vee \psi \wedge \theta = (\varphi \vee \psi) \wedge (\varphi \vee \theta)$
Absorption II	$\varphi \wedge (\varphi \vee \theta) = \varphi$	$\varphi \vee \varphi \wedge \theta = \varphi$
De Morgan's Laws	$\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$	$\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$
Double negation		$\neg\neg\varphi = \varphi$
Definitions	$\varphi \Rightarrow \psi = \neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$


©: Michael Kohlhase
219


We have seen in the examples above that while it is possible to get by with only the connectives \vee and \neg , it is a bit unnatural and tedious, since we need to eliminate the other connectives first. In this section, we will make the calculus less frugal by adding rules for the other connectives, without losing the advantage of dealing with a small calculus, which is good making statements about the calculus.

The main idea is to add the new rules as derived rules, i.e. inference rules that only abbreviate deductions in the original calculus. Generally, adding derived inference rules does not change the derivability relation of the calculus, and is therefore a safe thing to do. In particular, we will add the following rules to our tableau system.

We will convince ourselves that the first rule is a derived rule, and leave the other ones as an exercise.

Derived Rules of Inference

▷ **Definition 8.1.14** Let \mathcal{C} be a calculus, a rule of inference $\frac{A_1 \cdots A_n}{C}$ is called a **derived inference rule** in \mathcal{C} , iff there is a \mathcal{C} -proof of $A_1, \dots, A_n \vdash C$.

▷ **Definition 8.1.15** We have the following derived rules of inference

$$\begin{array}{c}
 \frac{A \Rightarrow B^t}{A^f \mid B^t} \quad \frac{A \Rightarrow B^f}{A^t \mid B^f} \quad \frac{A^t}{A \Rightarrow B^t} \quad \frac{A^t}{\neg A \vee B^t} \\
 \frac{A \vee B^t}{A^t \mid B^t} \quad \frac{A \vee B^f}{A^f \mid B^f} \quad \frac{A \Leftrightarrow B^t}{A^t \mid A^f \mid B^t} \quad \frac{A \Leftrightarrow B^f}{A^t \mid A^f \mid B^f} \quad \frac{A^t}{\neg(\neg\neg A \wedge \neg B)^t} \\
 \frac{A^f}{\neg\neg A \wedge \neg B^f} \quad \frac{A^f}{\neg\neg A^f \mid \neg B^f} \quad \frac{A^f}{\neg A^t \mid B^t} \quad \frac{A^f}{\perp}
 \end{array}$$

©:Michael Kohlhase 220

With these derived rules, theorem proving becomes quite efficient. With these rules, the tableau (?tab:firsttab?) would have the following simpler form:

Tableaux with derived Rules (example)

Example 8.1.16

$$\begin{array}{c}
 \text{love(mary, bill)} \wedge \text{love(john, mary)} \Rightarrow \text{love(john, mary)}^f \\
 \text{love(mary, bill)} \wedge \text{love(john, mary)}^t \\
 \text{love(john, mary)}^f \\
 \text{love(mary, bill)}^t \\
 \text{love(john, mary)}^t \\
 \perp
 \end{array}$$

©:Michael Kohlhase 221

EdN:4

Another thing that was awkward in (?tab:firsttab?) was that we used a proof for an implication to prove logical consequence. Such tests are necessary for instance, if we want to check consistency or informativity of new sentences⁴. Consider for instance a discourse $\Delta = D^1, \dots, D^n$, where n is large. To test whether a hypothesis \mathcal{H} is a consequence of Δ ($\Delta \models \mathcal{H}$) we need to show that $C := (D^1 \wedge \dots) \wedge D^n \Rightarrow \mathcal{H}$ is valid, which is quite tedious, since C is a rather large formula, e.g. if Δ is a 300 page novel. Moreover, if we want to test entailment of the form $(\Delta \models \mathcal{H})$ often, – for instance to test the informativity and consistency of every new sentence \mathcal{H} , then successive Δ s will overlap quite significantly, and we will be doing the same inferences all over again; the entailment check is not incremental.

EdN:5

Fortunately, it is very simple to get an incremental procedure for entailment checking in the model-generation-based setting: To test whether $\Delta \models \mathcal{H}$, where we have interpreted Δ in a model generation tableau \mathcal{T} , just check whether the tableau closes, if we add $\neg\mathcal{H}$ to the open branches. Indeed, if the tableau closes, then $\Delta \wedge \neg\mathcal{H}$ is unsatisfiable, so $\neg((\Delta \wedge \neg\mathcal{H}))$ is valid⁵, but this is equivalent to $\Delta \Rightarrow \mathcal{H}$, which is what we wanted to show.

Example 8.1.17 Consider for instance the following entailment in natural language.

Mary loves Bill. John loves Mary \models *John loves Mary*

⁴EdNOTE: add reference to presupposition stuff

⁵EdNOTE: Fix precedence of negation

⁶ We obtain the tableau

$$\begin{array}{c} \text{love}(\text{mary}, \text{bill})^t \\ \text{love}(\text{john}, \text{mary})^t \\ \neg(\text{love}(\text{john}, \text{mary}))^t \\ \text{love}(\text{john}, \text{mary})^f \\ \perp \end{array}$$

EdN:6

which shows us that the conjectured entailment relation really holds.

8.1.3 Soundness and Termination of Tableaux

As always we need to convince ourselves that the calculus is sound, otherwise, tableau proofs do not guarantee validity, which we are after. Since we are now in a refutation setting we cannot just show that the inference rules preserve validity: we care about unsatisfiability (which is the dual notion to validity), as we want to show the initial labeled formula to be unsatisfiable. Before we can do this, we have to ask ourselves, what it means to be (un)-satisfiable for a labeled formula or a tableau.

Soundness (Tableau)

- ▷ **Idea:** A test calculus is sound, iff it preserves satisfiability and the goal formulae are unsatisfiable.
- ▷ **Definition 8.1.18** A labeled formula \mathbf{A}^α is valid under φ , iff $\mathcal{I}_\varphi(\mathbf{A}) = \alpha$.
- ▷ **Definition 8.1.19** A tableau \mathcal{T} is satisfiable, iff there is a satisfiable branch \mathcal{P} in \mathcal{T} , i.e. if the set of formulae in \mathcal{P} is satisfiable.
- ▷ **Lemma 8.1.20** *Tableau rules transform satisfiable tableaux into satisfiable ones.*
- ▷ **Theorem 8.1.21 (Soundness)** *A set Φ of propositional formulae is valid, if there is a closed tableau \mathcal{T} for Φ^f .*
- ▷ **Proof:** by contradiction: Suppose Φ is not valid.
 - P.1** then the initial tableau is satisfiable (Φ^f satisfiable)
 - P.2** so \mathcal{T} is satisfiable, by Lemma 8.1.20.
 - P.3** there is a satisfiable branch (by definition)
 - P.4** but all branches are closed (\mathcal{T} closed)

□



Thus we only have to prove Lemma 8.1.20, this is relatively easy to do. For instance for the first rule: if we have a tableau that contains $\mathbf{A} \wedge \mathbf{B}^t$ and is satisfiable, then it must have a satisfiable branch. If $\mathbf{A} \wedge \mathbf{B}^t$ is not on this branch, the tableau extension will not change satisfiability, so we can assume that it is on the satisfiable branch and thus $\mathcal{I}_\varphi(\mathbf{A} \wedge \mathbf{B}) = \top$ for some variable assignment φ . Thus $\mathcal{I}_\varphi(\mathbf{A}) = \top$ and $\mathcal{I}_\varphi(\mathbf{B}) = \top$, so after the extension (which adds the formulae \mathbf{A}^t and \mathbf{B}^t to the branch), the branch is still satisfiable. The cases for the other rules are similar.

The next result is a very important one, it shows that there is a procedure (the tableau procedure) that will always terminate and answer the question whether a given propositional formula is valid

⁶EDNOTE: need to mark up the embedding of NL strings into Math

or not. This is very important, since other logics (like the often-studied first-order logic) does not enjoy this property.

Termination for Tableaux

▷ **Lemma 8.1.22** *The tableau procedure terminates, i.e. after a finite set of rule applications, it reaches a tableau, so that applying the tableau rules will only add labeled formulae that are already present on the branch.*

▷ Let us call a labeled formulae A^α **worked off** in a tableau \mathcal{T} , if a tableau rule has already been applied to it.

▷ **Proof:**

P.1 It is easy to see that applying rules to worked off formulae will only add formulae that are already present in its branch.

P.2 Let $\mu(\mathcal{T})$ be the number of connectives in a labeled formulae in \mathcal{T} that are not worked off.

P.3 Then each rule application to a labeled formula in \mathcal{T} that is not worked off reduces $\mu(\mathcal{T})$ by at least one. (inspect the rules)

P.4 at some point the tableau only contains worked off formulae and literals.

P.5 since there are only finitely many literals in \mathcal{T} , so we can only apply the tableau cut rule a finite number of times. \square



The Tableau calculus basically computes the disjunctive normal form: every branch is a disjunct that is a conjunct of literals. The method relies on the fact that a DNF is unsatisfiable, iff each monomial is, i.e. iff each branch contains a contradiction in form of a pair of complementary literals.

8.2 Resolution for Propositional Logic

The next calculus is a test calculus based on the conjunctive normal form. In contrast to the tableau method, it does not compute the normal form as it goes along, but has a pre-processing step that does this and a single inference rule that maintains the normal form. The goal of this calculus is to derive the empty clause (the empty disjunction), which is unsatisfiable.

Another Test Calculus: Resolution

▷ **Definition 8.2.1** A **clause** is a disjunction of literals. We will use \square for the empty disjunction (no disjuncts) and call it the **empty clause**.

▷ **Definition 8.2.2 (Resolution Calculus)** The **resolution calculus** operates a clause sets via a single inference rule:

$$\frac{P^t \vee \mathbf{A} \quad P^f \vee \mathbf{B}}{\mathbf{A} \vee \mathbf{B}}$$

This rule allows to add the clause below the line to a clause set which contains the two clauses above.

▷ **Definition 8.2.3 (Resolution Refutation)** Let S be a clause set, and $D: S \vdash_{\mathcal{R}} T$ a \mathcal{R} derivation then we call D **resolution refutation**, iff $\square \in T$.



©: Michael Kohlhase

224



A calculus for CNF Transformation

▷ **Definition 8.2.4 (Transformation into Conjunctive Normal Form)**
The **CNF transformation calculus** \mathcal{CNF} consists of the following four inference rules on clause sets.

$$\frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^t}{\mathbf{C} \vee \mathbf{A}^t \vee \mathbf{B}^t} \quad \frac{\mathbf{C} \vee (\mathbf{A} \vee \mathbf{B})^f}{\mathbf{C} \vee \mathbf{A}^f; \mathbf{C} \vee \mathbf{B}^f} \quad \frac{\mathbf{C} \vee \neg \mathbf{A}^t}{\mathbf{C} \vee \mathbf{A}^f} \quad \frac{\mathbf{C} \vee \neg \mathbf{A}^f}{\mathbf{C} \vee \mathbf{A}^t}$$

▷ **Definition 8.2.5** We write $CNF(\mathbf{A})$ for the set of all clauses derivable from \mathbf{A}^f via the rules above.

▷ **Definition 8.2.6 (Resolution Proof)** We call a resolution refutation $\mathcal{P}: CNF(\mathbf{A}) \vdash_{\mathcal{R}} T$ a **resolution proof** for $\mathbf{A} \in \text{wff}_o(\mathcal{V}_o)$.



©: Michael Kohlhase

225



Note: Note that the \mathbf{C} -terms in the definition of the resolution calculus are necessary, since we assumed that the assumptions of the inference rule must match full formulae. The \mathbf{C} -terms are used with the convention that they are optional. So that we can also simplify $(\mathbf{A} \vee \mathbf{B})^t$ to $\mathbf{A}^t \vee \mathbf{B}^t$.

The background behind this notation is that \mathbf{A} and $T \vee \mathbf{A}$ are equivalent for any \mathbf{A} . That allows us to interpret the \mathbf{C} -terms in the assumptions as T and thus leave them out.

The resolution calculus as we have formulated it here is quite frugal; we have left out rules for the connectives \vee , \Rightarrow , and \Leftrightarrow , relying on the fact that formulae containing these connectives can be translated into ones without before CNF transformation. The advantage of having a calculus with few inference rules is that we can prove meta-properties like soundness and completeness with less effort (these proofs usually require one case per inference rule). On the other hand, adding specialized inference rules makes proofs shorter and more readable.

Fortunately, there is a way to have your cake and eat it. Derived inference rules have the property that they are formally redundant, since they do not change the expressive power of the calculus. Therefore we can leave them out when proving meta-properties, but include them when actually using the calculus.

Derived Rules of Inference

▷ **Definition 8.2.7** Let \mathcal{C} be a calculus, a rule of inference $\frac{\mathbf{A}_1 \quad \dots \quad \mathbf{A}_n}{\mathbf{C}}$ is



called a **derived inference rule** in \mathcal{C} , iff there is a \mathcal{C} -proof of $\mathbf{A}_1, \dots, \mathbf{A}_n \vdash \mathbf{C}$.

▷ **Example 8.2.8**

$$\frac{\frac{\frac{C \vee (A \Rightarrow B)^t}{C \vee (\neg A \vee B)^t}}{C \vee \neg A^t \vee B^t}}{C \vee A^f \vee B^t} \mapsto \frac{C \vee (A \Rightarrow B)^t}{C \vee A^f \vee B^t}$$

▷ **Others:**

$$\frac{C \vee (A \Rightarrow B)^t}{C \vee A^f \vee B^t} \quad \frac{C \vee (A \Rightarrow B)^f}{C \vee A^t; C \vee B^f} \quad \frac{C \vee A \wedge B^t}{C \vee A^t; C \vee B^t} \quad \frac{C \vee A \wedge B^f}{C \vee A^f \vee B^f}$$


©:Michael Kohlhase
226


With these derived rules, theorem proving becomes quite efficient. To get a better understanding of the calculus, we look at an example: we prove an axiom of the Hilbert Calculus we have studied above.

Example: Proving Axiom S



▷ **Example 8.2.9** Clause Normal Form transformation

$$\frac{\frac{(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R^f}{P \Rightarrow Q \Rightarrow R^t; (P \Rightarrow Q) \Rightarrow P \Rightarrow R^f}}{P^f \vee (Q \Rightarrow R)^t; P \Rightarrow Q^t; P \Rightarrow R^f}}{P^f \vee Q^f \vee R^t; P^f \vee Q^t; P^t; R^f}$$

$CNF = \{P^f \vee Q^f \vee R^t, P^f \vee Q^t, P^t, R^f\}$

▷ **Example 8.2.10** Resolution Proof

1	$P^f \vee Q^f \vee R^t$	initial
2	$P^f \vee Q^t$	initial
3	P^t	initial
4	R^f	initial
5	$P^f \vee Q^f$	resolve 1.3 with 4.1
6	Q^f	resolve 5.1 with 3.1
7	P^f	resolve 2.2 with 6.1
8	□	resolve 7.1 with 3.1


©:Michael Kohlhase
227


Part II

Interlude for the Semester Change

Chapter 9

Legal Foundations of Information Technology

In this chapter, we cover a topic that is a very important secondary aspect of our work as Computer Scientists: the legal foundations that regulate how the fruits of our labor are appreciated (and recompensated), and what we have to do to respect people's personal data.

9.1 Intellectual Property, Copyright, and Licensing

The first complex of questions centers around the assessment of the products of work of knowledge/information workers, which are largely intangible, and about questions of recompensation for such work.

Intellectual Property: Concept

- ▷ **Question:** Intellectual labour creates (intangible) objects, can they be owned?
- ▷ **Answer:** Yes: in certain circumstances they are property like tangible objects.
- ▷ **Definition 9.1.1** The concept of **intellectual property** motivates a set of laws that regulate property rights on intangible objects, in particular
 - ▷ **Patents** grant exploitation rights on original ideas.
 - ▷ **Copyrights** grant personal and exploitation rights on expressions of ideas.
 - ▷ **Industrial Design Rights** protect the visual design of objects beyond their function.
 - ▷ **Trademarks** protect the signs that identify a legal entity or its products to establish brand recognition.
- ▷ **Intent:** Property-like treatment of intangibles will foster innovation by giving individuals and organizations material incentives.



Naturally, many of the concepts are hotly debated. Especially due to the fact that intuitions and legal systems about property have evolved around the more tangible forms of properties

that cannot be simply duplicated and indeed multiplied by copying them. In particular, other intangibles like physical laws or mathematical theorems cannot be property.

Intellectual Property: Problems

- ▷ **Delineation Problems:** How can we distinguish the product of human work, from “discoveries”, of e.g. algorithms, facts, genome, algorithms. (not property)
- ▷ **Philosophical Problems:** The implied analogy with physical property (like land or an automobile) fails because physical property is generally rivalrous while intellectual works are non-rivalrous (the enjoyment of the copy does not prevent enjoyment of the original).
- ▷ **Practical Problems:** There is widespread criticism of the concept of intellectual property in general and the respective laws in particular.
 - ▷ (software) patents are often used to stifle innovation in practice. (patent trolls)
 - ▷ copyright is seen to help big corporations and to hurt the innovating individuals



We will not go into the philosophical debates around intellectual property here, but concentrate on the legal foundations that are in force now and regulate IP issues. We will see that groups holding alternative views of intellectual properties have learned to use current IP laws to their advantage and have built systems and even whole sections of the software economy on this basis.

Many of the concepts we will discuss here are regulated by laws, which are (ultimately) subject to national legislative and juridicative systems. Therefore, none of them can be discussed without an understanding of the different jurisdictions. Of course, we cannot go into particulars here, therefore we will make use of the classification of jurisdictions into two large legal traditions to get an overview. For any concrete decisions, the details of the particular jurisdiction have to be checked.

Legal Traditions

- ▷ The various legal systems of the world can be grouped into “traditions”.
- ▷ **Definition 9.1.2** Legal systems in the **common law tradition** are usually based on case law, they are often derived from the British system.
- ▷ **Definition 9.1.3** Legal systems in the **civil law tradition** are usually based on explicitly codified laws (civil codes).
- ▷ As a rule of thumb all English-speaking countries have systems in the **common law tradition**, whereas the rest of the world follows a **civil law tradition**.



Another prerequisite for understanding intellectual property concepts is the historical development of the legal frameworks and the practice how intellectual property law is synchronized internationally.

Historic/International Aspects of Intellectual Property Law

- ▷ **Early History:** In **late antiquity** and the **middle ages** IP matters were regulated by royal privileges
- ▷ **History of Patent Laws:** First in Venice 1474, Statutes of Monopolies in England 1624, US/France 1790/1...
- ▷ **History of Copyright Laws:** Statue of Anne 1762, France: 1793, ...
- ▷ **Problem:** In an increasingly globalized world, national IP laws are not enough.
- ▷ **Definition 9.1.4** The **Berne convention** process is a series of international treaties that try to harmonize international IP laws. It started with the original Berne convention 1886 and went through revision in 1896, 1908, 1914, 1928, 1948, 1967, 1971, and 1979.
- ▷ The World Intellectual Property Organization Copyright Treaty was adopted in 1996 to address the issues raised by information technology and the Internet, which were not addressed by the Berne Convention.
- ▷ **Definition 9.1.5** The **Anti-Counterfeiting Trade Agreement (ACTA)** is a multinational treaty on international standards for intellectual property rights enforcement.
- ▷ With its focus on enforcement **ACTA** is seen by many to break fundamental human information rights, criminalize FLOSS



9.1.1 Copyright

In this subsection, we go into more detail about a central concept of **intellectual property** law: copyright is the component most of IP law applicable to the individual computer scientist. Therefore a basic understanding should be part of any CS education. We start with a definition of what works can be copyrighted, and then progress to the rights this affords to the copyright holder.

Copyrightable Works

- ▷ **Definition 9.1.6** A **copyrightable work** is any artefact of human labor that fits into one of the following eight categories:
 - ▷ **Literary work** s: Any work expressed in letters, numbers, or symbols, regardless of medium. (Computer source code is also considered to be a literary work.)
 - ▷ **Musical works:** Original musical compositions.
 - ▷ **Sound recording** s of musical works. (different licensing)
 - ▷ **Dramatic works:** literary works that direct a performance through written instructions.
 - ▷ **Choreographic works** must be fixed, either through notation or video recording.

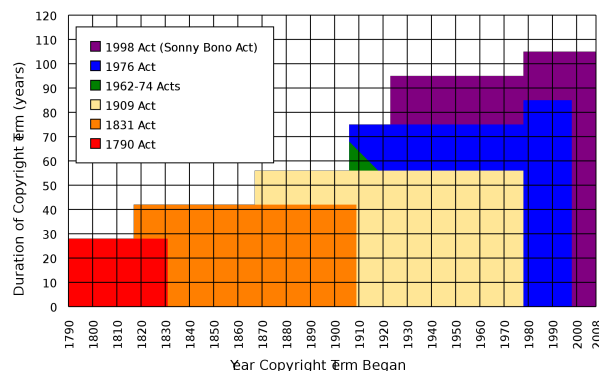
- ▷ **Pictorial, Graphic and Sculptural works (PGS works)**: Any two-dimensional or three-dimensional art work
- ▷ **Audiovisual works**: work that combines audio and visual components. (e.g. films, television programs)
- ▷ **Architectural works** (copyright only extends to aesthetics)
- ▷ The categories are interpreted quite liberally (e.g. for computer code).
- ▷ There are various requirements to make a work copyrightable: it has to
 - ▷ exhibit a certain originality (Schöpfungshöhe)
 - ▷ require a certain amount of labor and diligence ("sweat of the brow" doctrine)



In short almost all products of intellectual work are copyrightable, but this does not mean copyright applies to all those works. Indeed there is a large body of works that are “out of copyright”, and can be used by everyone. Indeed it is one of the intentions of **intellectual property** laws to increase the body of intellectual resources a society can draw upon to create wealth. Therefore copyright is limited by regulations that limit the duration of copyright and exempts some classes of works from copyright (e.g. because they have already been paid for by society).

Limitations of Copyrightability: The Public Domain

- ▷ **Definition 9.1.7** A work is said to be in the **public domain**, if no copyright applies, otherwise it is called **copyrighted**.
- ▷ **Example 9.1.8** Works made by US government employees (in their work time) are in the public domain directly (Rationale: taxpayer already paid for them)
- ▷ **Copyright expires**: usually 70 years after the death of the creator
- ▷ **Example 9.1.9 (US Copyright Terms)** Some people claim that US copyright terms are extended, whenever Disney's Mickey Mouse would become **public domain**.



Now that we have established, which works are copyrighted — i.e. to which works are [intellectual property](#), let us see who owns them, and how that ownership is established.

Copyright Holder

- ▷ **Definition 9.1.10** The **copyright holder** is the legal entity that holds the **copyright** to a **copyrighted** work.
- ▷ By default, the original creator of a copyrightable work holds the copyright.
- ▷ In most jurisdictions, no registration or declaration is necessary (**but copyright ownership may be difficult to prove**)
- ▷ copyright is considered **intellectual property**, and can be transferred to others (e.g. **sold to a publisher or bequeathed**)
- ▷ **Definition 9.1.11 (Work for Hire)** A **work made for hire** is a work created by an employee as part of his or her job, or under the explicit guidance or under the terms of a contract.
- ▷ *In jurisdictions from the **common law tradition**, the copyright holder of a **work for hire** is the employer, in jurisdictions from the **civil law tradition**, the author, unless the respective contract regulates it otherwise.*



We now turn to the rights owning a copyright entails for the copyright holder.

Rights under Copyright Law

- ▷ **Definition 9.1.12** The **copyright** is a collection of rights on a **copyrighted** work;
 - ▷ **personal rights**: the copyright holder may
 - ▷ determine whether and how the work is published (**right to publish**)
 - ▷ determine whether and how her authorship is acknowledged. (**right of attribution**)
 - ▷ to object to any distortion, mutilation or other modification of the work, which would be prejudicial to his honor or reputation (**droit de respect**)
 - ▷ **exploitation rights**: the owner of a copyright has the exclusive right to do, or authorize to do any of the following:
 - ▷ to reproduce the copyrighted work in copies (or phonorecords);
 - ▷ to prepare derivative works based upon the copyrighted work;
 - ▷ to distribute copies of the work to the public by sale, rental, lease, or lending;
 - ▷ to perform the copyrighted work publicly;
 - ▷ to display the copyrighted work publicly; and
 - ▷ to perform the copyrighted work publicly by means of a digital-audio transmission.
- ▷ **Definition 9.1.13** The use of a copyrighted material, by anyone other than the owner of the copyright, amounts to **copyright infringement** only when the

use is such that it conflicts with any one or more of the exclusive rights conferred to the owner of the copyright.



©: Michael Kohlhase

235



Again, the rights of the copyright holder are mediated by usage rights of society; recall that [intellectual property](#) laws are originally designed to increase the intellectual resources available to society.

Limitations of Copyright (Citation/Fair Use)

- ▷ There are limitations to the exclusivity of rights of the [copyright holder](#) (some things cannot be forbidden)
- ▷ **Citation Rights:** [Civil law jurisdictions](#) allow citations of (extracts of) copyrighted works for scientific or artistic discussions. (note that the right of attribution still applies)
- ▷ In the [civil law tradition](#), there are similar rights:
- ▷ **Definition 9.1.14 (Fair Use/Fair Dealing Doctrines)** Case law in [common law jurisdictions](#) has established a [fair use doctrine](#), which allows e.g.
 - ▷ making safety copies of software and audiovisual data
 - ▷ lending of books in public libraries
 - ▷ citing for scientific and educational purposes
 - ▷ excerpts in search engine

Fair use is established in court on a case-by-case taking into account the purpose (commercial/educational), the nature of the work the amount of the excerpt, the effect on the marketability of the work.



©: Michael Kohlhase

236



9.1.2 Licensing

Given that [intellectual property](#) law grants a set of exclusive rights to the owner, we will now look at ways and mechanisms how usage rights can be bestowed on others. This process is called licensing, and it has enormous effects on the way software is produced, marketed, and consumed. Again, we will focus on copyright issues and how innovative license agreements have created the open source movement and economy.

Licensing: the Transfer of Rights

- ▷ **Remember:** the [copyright holder](#) has [exclusive rights](#) to a [copyrighted](#) work.
- ▷ **In particular:** all others have only [fair-use rights](#) (but we can transfer rights)
- ▷ **Definition 9.1.15** A [license](#) is an authorization (by the [licensor](#)) to use the licensed material (by the [licensee](#)).

- ▷ **Note:** a **license** is a regular contract (about **intellectual property**) that is handled just like any other contract. (it can stipulate anything the licensor and licensees agree on) in particular a license may
 - ▷ involve **term, territory, or renewal** provisions
 - ▷ require paying a fee and/or proving a capability.
 - ▷ require to keep the licensor informed on a type of activity, and to give them the opportunity to set conditions and limitations.
- ▷ **Mass Licensing of Computer Software:** Software vendors usually license software under extensive **end-user license agreement** (EULA) entered into upon the installation of that software on a computer. The license authorizes the user to install the software on a limited number of computers.



Copyright law was originally designed to give authors of literary works — e.g. novelists and playwrights — revenue streams and regulate how publishers and theatre companies can distribute and display them so that society can enjoy more of their work.

With the inclusion of software as “literary works” under copyright law the basic parameters of the system changed considerably:

- modern software development is much more a collaborative and diversified effort than literary writing,
- re-use of software components is a decisive factor in software,
- software can be distributed in compiled form to be executable which limits inspection and re-use, and
- distribution costs for digital media are negligible compared to printing.

As a consequence, much software development has been industrialized by large enterprises, who become **copyrights** the software was created as **work for hire** This has led to software quasi-monopolies, which are prone to stifling innovation and thus counteract the intentions of intellectual property laws.

The **Free/Open Source Software** movement attempts to use the **intellectual property** laws themselves to counteract their negative side effects on innovation and collaboration and the (perceived) freedom of the programmer.

Free/Open Source Licenses

- ▷ **Recall:** Software is treated as literary works wrt. copyright law.
- ▷ **But:** Software is different from literary works wrt. distribution channels (and that is what copyright law regulates)
- ▷ **In particular:** When literary works are distributed, you get all there is, software is usually distributed in binary format, you cannot understand/cite/modify/fix it.
- ▷ **So:** Compilation can be seen as a technical means to enforce copyright. (seen as an impediment to freedom of fair use)
- ▷ **Recall:** IP laws (in particular patent law) was introduced explicitly for two things

- ▷ incentivize innovation (by granting exclusive exploitation rights)
 - ▷ spread innovation (by publishing ideas and processes)
- Compilation breaks the second tenet (and may thus stifle innovation)

- ▷ **Idea:** We should create a public domain of source code
- ▷ **Definition 9.1.16** **Free/Libre/Open-Source Software** (FLOSS) is software that is and licensed via licenses that ensure that its source is available.
- ▷ Almost all of the Internet infrastructure is (now) FLOSS; so are the Linux and Android operating systems and applications like OpenOffice and The GIMP.



©:Michael Kohlhase

238



The relatively complex name **Free/Libre/Open Source** comes from the fact that the English¹ word “free” has two meanings: free as in “freedom” and free as in “free beer”. The initial name “free software” confused issues and thus led to problems in public perception of the movement. Indeed Richard Stallman’s initial motivation was to ensure the freedom of the programmer to create software, and only used cost-free software to expand the software public domain. To disambiguate some people started using the French “libre” which only had the “freedom” reading of “free”. The term “open source” was eventually adopted in 1998 to have a politically less loaded label.



The main tool in brining about a **public domain** of **open-source software** was the use of licenses that are cleverly crafted to guarantee usage rights to the public and inspire programmers to license their works as open-source systems. The most influential license here is the Gnu public license which we cover as a paradigmatic example.

GPL/Copyleft: Creating a FLOSS Public Domain?

- ▷ **Problem:** How do we get people to contribute source code to the **FLOSS** public domain?
- ▷ **Idea:** Use special licenses to:
 - ▷ allow others to use/fix/modify our source code (derivative works)
 - ▷ require them to release their modifications to the **FLOSS** public domain if they do.
- ▷ **Definition 9.1.17** A **copyleft** license is a license which requires that allows derivative works, but requires that they be licensed with the same license.
- ▷ **Definition 9.1.18** The **General Public License** (GPL) is a **copyleft** license for **FLOSS** software originally written by Richard Stallman in 1989. It requires that the source code of GPL-licensed software be made available.
- ▷ The GPL was the first copyleft license to see extensive use, and continues to dominate the licensing of **FLOSS** software.
- ▷ **FLOSS** based development can reduce development and testing costs (but **community involvement must be managed**)

¹the movement originated in the USA

▷ Various software companies have developed successful business models based on **FLOSS** licensing models. (e.g. Red Hat, Mozilla, IBM, ...)

 ©: Michael Kohlhase 239 

Note: that the GPL does not make any restrictions on possible uses of the software. In particular, it does not restrict commercial use of the copyrighted software. Indeed it tries to allow commercial use without restricting the freedom of programmers. If the unencumbered distribution of source code makes some business models (which are considered as “extortion” by the open-source proponents) intractable, this needs to be compensated by new, innovative business models. Indeed, such business models have been developed, and have led to an “open-source economy” which now constitutes a non-trivial part of the software industry.

With the great success of **open-source software**, the central ideas have been adapted to other classes of copyrightable works; again to create and enlarge a public domain of resources that allow re-use, derived works, and distribution.


Open Content via Open Content Licenses

▷ **Recall:** **FLOSS** licenses have created a vibrant public domain for software.

▷ **How about:** other copyrightable works: music, video, literature, technical documents



Definition 9.1.19 The **Creative Commons license s** are

- ▷ a **common legal vocabulary** for sharing content
- ▷ to create a kind of “public domain” using licensing
- ▷ presented in three layers (human/lawyer/machine)-readable



▷ Creative Commons license provisions (<http://www.creativecommons.org>)

- ▷ **author retains copyright** on each module/course
- ▷ **author licenses** material to the world with requirements
 - +/- **attribution** (must reference the author)
 - +/- **commercial use** (can be restricted)
 - +/- **derivative works** (can allow modification)
 - +/- **share alike (copyleft)** (modifications must be donated back)

 ©: Michael Kohlhase 240 

9.2 Information Privacy

Information/Data Privacy

▷ **Definition 9.2.1** The principle of **information privacy** comprises the idea that humans have the right to control who can access their personal data when.

- ▷ Information privacy concerns exist wherever personally identifiable information is collected and stored – in digital form or otherwise. In particular in the following contexts
 - ▷ Healthcare records
 - ▷ Criminal justice investigations and proceedings
 - ▷ Financial institutions and transactions
 - ▷ Biological traits, such as ethnicity or genetic material
 - ▷ Residence and geographic records
- ▷ Information privacy is becoming a growing concern with the advent of the Internet and search engines that make access to information easy and efficient.
- ▷ The “reasonable expectation of privacy” is regulated by special laws.
- ▷ These laws differ considerably by jurisdiction; Germany has particularly stringent regulations (and you are subject to these.)

Acquisition and storage of personal data is only legal for the purposes of the respective transaction, must be minimized, and distribution of personal data is generally forbidden with few exceptions. Users have to be informed about collection of personal data.



Organizational Measures or Information Privacy (under German Law)

- ▷ **Physical Access Control:** Unauthorized persons may not be granted physical access to data processing equipment that process personal data. (↪ locks, access control systems)
- ▷ **System Access Control:** Unauthorized users may not use systems that process personal data (↪ passwords, firewalls, ...)
- ▷ **Information Access Control:** Users may only access those data they are authorized to access. (↪ access control lists, safe boxes for storage media, encryption)
- ▷ **Data Transfer Control:** Personal data may not be copied during transmission between systems (↪ encryption)
- ▷ **Input Control:** It must be possible to review retroactively who entered, changed, or deleted personal data. (↪ authentication, journaling)
- ▷ **Availability Control:** Personal data have to be protected against loss and accidental destruction (↪ physical/building safety, backups)
- ▷ **Obligation of Separation:** Personal data that was acquired for separate purposes has to be processed separately.



Chapter 10

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!)



©: Michael Kohlhase

243



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 Computer Scientists)
 - ▷ *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?)



©: Michael Kohlhase

244



10.1 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



©:Michael Kohlhase

246



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 11

Combinational Circuits

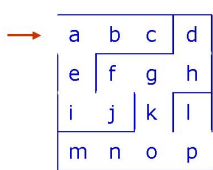
We will now study a new model of computation that comes quite close to the circuits that execute computation on today's computers. Since the course studies computation in the context of computer science, we will abstract away from all physical issues of circuits, in particular the construction of gates and timing issues. This allows 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 Section 11.0, and then build a simple theory of combinational circuits in Section 11.1 and study their time and space complexity in Section 11.2. We introduce combinational circuits for computing with numbers, by introducing positional number systems and addition in Section 12.0 and covering 2s-complement numbers and subtraction in Section 12.1. A basic introduction to sequential logic circuits and memory elements in Chapter 12 concludes our study of circuits.

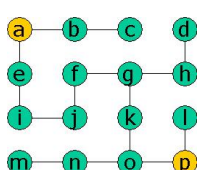
11.1 Graphs and Trees

Some more Discrete Math: Graphs and Trees

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



→




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

▷ We represented the maze as a graph for clarity.


▷ Now, we are interested in circuits, which we will also represent as graphs.

▷ Let us look at the theory of graphs first (so we know what we are doing)

 SOME RIGHTS RESERVED

©: Michael Kohlhase

247

 JACOBS UNIVERSITY

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 11.1.1** 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 11.1.2** 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 11.1.3** Given a graph $G = \langle V, E \rangle$. The **in-degree** $\text{indeg}(v)$ and the **out-degree** $\text{outdeg}(v)$ of a **vertex** $v \in V$ are defined as
 - ▷ $\text{indeg}(v) = \#\{\{w \mid \langle w, v \rangle \in E\}$
 - ▷ $\text{outdeg}(v) = \#\{\{w \mid \langle v, w \rangle \in E\}$

Note: For an undirected graph, $\text{indeg}(v) = \text{outdeg}(v)$ for all nodes v .



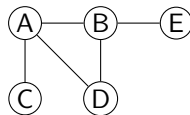
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for indeg and outdeg are the obvious variants: $\text{indeg}(v)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

In the following if we do not specify that a graph is undirected, it will be assumed to be directed.

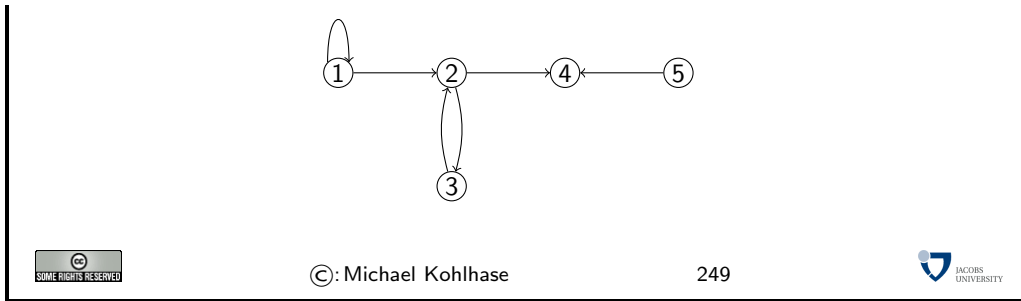
This is a very abstract yet elementary definition. We only need very basic concepts like sets and ordered pairs to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

▷ Examples

- ▷ **Example 11.1.4** 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 11.1.5** A directed graph $G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 2, 4 \rangle, \langle 5, 4 \rangle\}$



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 250

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.

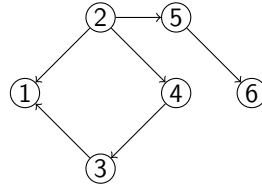
Directed Graphs

- ▷ **Idea:** Directed Graphs are nothing else than relations
- ▷ **Definition 11.1.6** Let $G = \langle V, E \rangle$ be a **directed graph**, then we call a **node** $v \in V$
 - ▷ **initial**, iff there is no $w \in V$ such that $\langle w, v \rangle \in E$. (no predecessor)

▷ **terminal**, iff there is no $w \in V$ such that $\langle v, w \rangle \in E$. (no successor)

In a graph G , node v is also called a **source (sink)** of G , iff it is initial (terminal) in G .

▷ **Example 11.1.7** 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.

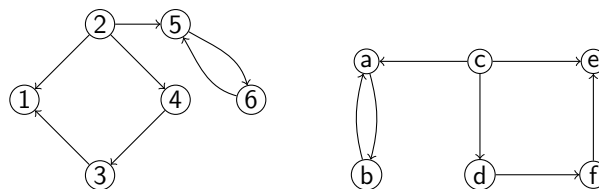
Graph Isomorphisms

▷ **Definition 11.1.8** A **graph isomorphism** between two graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is a bijective function $\psi: V \rightarrow V'$ with

directed graphs	undirected graphs
$\langle a, b \rangle \in E \Leftrightarrow \langle \psi(a), \psi(b) \rangle \in E'$	$\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in E'$

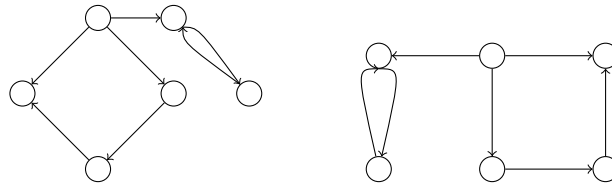
▷ **Definition 11.1.9** Two graphs G and G' are **equivalent** iff there is a graph-isomorphism ψ between G and G' .

▷ **Example 11.1.10** 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 priori 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 11.1.11** A **labeled graph** G is a triple $\langle V, E, f \rangle$ where $\langle V, E \rangle$ is a graph and $f: V \cup E \rightarrow R$ is a partial function into a set R of **label** s.
- ▷ **Notation 11.1.12** 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 11.1.13** $G = \langle V, E, f \rangle$ with $V = \{A, B, C, D, E\}$, where
 - ▷ $E = \{\langle A, A \rangle, \langle A, B \rangle, \langle B, C \rangle, \langle C, B \rangle, \langle B, D \rangle, \langle E, D \rangle\}$
 - ▷ $f: V \cup E \rightarrow \{+, -, \emptyset\} \times \{1, \dots, 9\}$ with
 - ▷ $f(A) = 5, f(B) = 3, f(C) = 7, f(D) = 4, f(E) = 8,$
 - ▷ $f(\langle A, A \rangle) = -0, f(\langle A, B \rangle) = -2, f(\langle B, C \rangle) = +4,$
 - ▷ $f(\langle C, B \rangle) = -4, f(\langle B, D \rangle) = +1, f(\langle E, D \rangle) = -4$

©: Michael Kohlhase

253

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 11.1.14** Given a directed graph $G = \langle V, E \rangle$, then we call a vector $p = \langle v_0, \dots, v_n \rangle \in V^{n+1}$ 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 $\text{len}(p)$)



Note: Not all v_i -s in a path are necessarily different.

▷ **Notation 11.1.15** For a graph $G = \langle V, E \rangle$ and a path $p = \langle v_0, \dots, v_n \rangle \in V^{n+1}$, 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 11.1.16** We write $\Pi(G)$ for the set of all paths in a graph G .


©: Michael Kohlhase
254


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

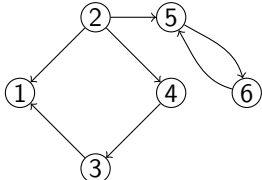
▷ **Definition 11.1.17** Given a graph $G = \langle V, E \rangle$, then

▷ a path p is called **cyclic** (or a **cycle**) iff $\text{start}(p) = \text{end}(p)$.

▷ a cycle $\langle v_0, \dots, v_n \rangle$ 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 11.1.18** $\langle 2, 4, 3 \rangle$ 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 11.1.19** We will sometimes use the abbreviation **DAG** for “**di**-**rected acyclic graph**”.


©: Michael Kohlhase
255


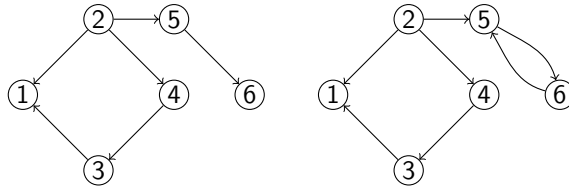
Of course, speaking about cycles is only meaningful in directed graphs, since undirected graphs can only be acyclic, iff they do not have edges at all.

Graph Depth

▷ **Definition 11.1.20** Let $G := \langle V, E \rangle$ be a digraph, then the **depth** $\text{dp}(v)$ of a vertex $v \in V$ is defined to be 0, if v is a source of G and $\sup(\{\text{len}(p) \mid \text{indeg}(\text{start}(p)) = 0 \wedge \text{end}(p) = v\})$ otherwise, i.e. the length of the longest path from a source of G to v . (⚠ can be infinite)

▷ **Definition 11.1.21** Given a digraph $G = \langle V, E \rangle$. The **depth** ($\text{dp}(G)$) of G is defined as $\sup(\{\text{len}(p) \mid p \in \Pi(G)\})$, i.e. the maximal path length in G .

▷ **Example 11.1.22** The vertex 6 has depth two in the left graph and infinite 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.

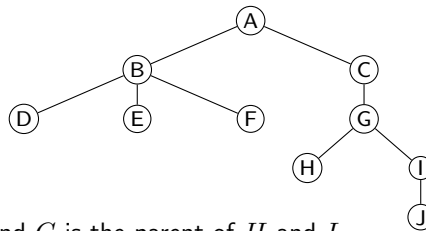
Trees

▷ **Definition 11.1.23** 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 11.1.24** 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 11.1.25** For any node $v \in V$ except the root v_r , there is exactly one path $p \in \Pi(G)$ with $\text{start}(p) = v_r$ and $\text{end}(p) = v$. (*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.

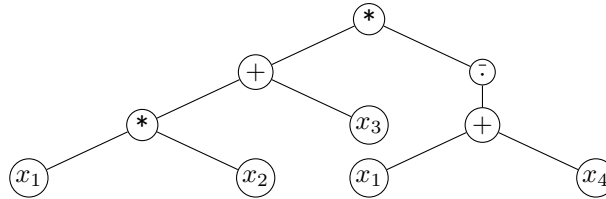
This allows us to view Boolean expressions as trees and apply all the mathematics (nomenclature and results) we will develop for them.

The Parse-Tree of a Boolean Expression

▷ **Definition 11.1.26** 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 $e = \bar{e}'$ then $V_e := V_{e'} \cup \{v\}$, $E_e := E_{e'} \cup \{\langle v, v'_r \rangle\}$, and $f_e := f_{e'} \cup \{v \mapsto \bar{\cdot}\}$, 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 $V_e := V_{e_1} \cup V_{e_2} \cup \{v\}$, $E_e := E_{e_1} \cup E_{e_2} \cup \{\langle v, v'_1 \rangle, \langle v, v'_2 \rangle\}$, 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 is the root of P_{e_i} and v is an object not in $V_{e_1} \cup V_{e_2}$.
- ▷ if $e \in (V \cup C_{\text{bool}})$ then, $V_e = \{e\}$ and $E_e = \emptyset$.

▷ **Example 11.1.27** the parse tree of $(x_1 * x_2 + x_3) * x_1 + x_4$ is



©: Michael Kohlhase

258



11.2 Introduction to Combinational Circuits

We will now come to another model of computation: combinational circuits. These are models of logic circuits (physical objects made of transistors (or cathode tubes) and wires, parts of integrated circuits, etc), which abstract from the inner structure for the switching elements (called gates) and the geometric configuration of the connections. Thus, combinational circuits allow us to concentrate on the functional properties of these circuits, without getting bogged down with e.g. configuration- or geometric considerations. These can be added to the models, but are not part of the discussion of this course.

Combinational Circuits as Graphs

▷ **Definition 11.2.1** A **combinational circuit** is a labeled acyclic graph $G = \langle V, E, f_g \rangle$ with label set $\{\text{OR}, \text{AND}, \text{NOT}\}$, such that

- ▷ $\text{indeg}(v) = 2$ and $\text{outdeg}(v) = 1$ for all nodes $v \in f_g^{-1}(\{\text{AND}, \text{OR}\})$
- ▷ $\text{indeg}(v) = \text{outdeg}(v) = 1$ for all nodes $v \in f_g^{-1}(\{\text{NOT}\})$

We call the set $I(G)$ ($O(G)$) of initial (terminal) nodes in G the **input** (**output**) vertices, and the set $F(G) := V \setminus (I(G) \cup O(G))$ the set of **gate** s.

▷ **Example 11.2.2** The following graph $G_{\text{cir1}} = \langle V, E \rangle$ is a combinational circuit

▷ **Definition 11.2.3** 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)

©: Michael Kohlhase 259

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)

©: Michael Kohlhase 260

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).

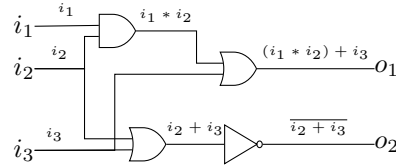
Computing with Combinational Circuits

- ▷ Combinational Circuits and parse trees for Boolean expressions look similar
- ▷ **Idea:** Let's annotate edges in combinational circuit with Boolean Expressions!

- ▷ **Definition 11.2.4** 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	$v \in I(G)$
$\overline{f_L(\langle u, v \rangle)}$	$f_g(v) = \text{NOT}$
$f_L(\langle u, v \rangle) * f_L(\langle u', v \rangle)$	$f_g(v) = \text{AND}$
$f_L(\langle u, v \rangle) + f_L(\langle u', v \rangle)$	$f_g(v) = \text{OR}$

▷ Example 11.2.5



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 11.2.6** A combinational circuit $G = \langle V, E, f_g \rangle$ with input vertices i_1, \dots, i_n and output vertices o_1, \dots, o_m **computes** an n -ary Boolean function

$$f: \{0, 1\}^n \rightarrow \{0, 1\}^m; \langle i_1, \dots, i_n \rangle \mapsto \langle f_{e_1}(i_1, \dots, i_n), \dots, f_{e_m}(i_1, \dots, i_n) \rangle$$

where $e_i = f_L(\langle v, o_i \rangle)$.

- ▷ **Example 11.2.7** The circuit in Example 11.2.5 computes the Boolean function $f: \{0, 1\}^3 \rightarrow \{0, 1\}^2; \langle i_1, i_2, i_3 \rangle \mapsto \langle f_{i_1 * i_2 + i_3}, f_{\overline{i_2 + i_3}} \rangle$
- ▷ **Definition 11.2.8** 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 .

11.3 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 (almost) the speed of light, gates have finite computation times.

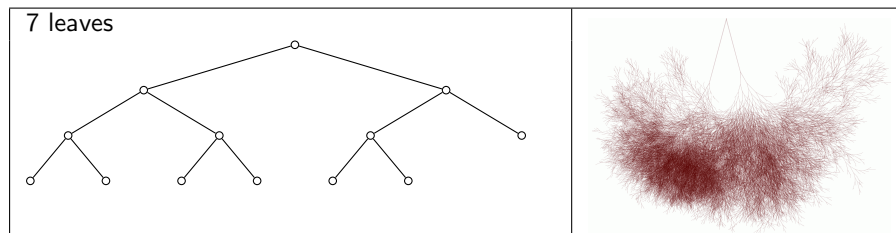
Therefore we look at special configurations for combinational circuits that have good depth and cost. These will become important, when we build actual combinational circuits with given input/output behavior. We will now study the case for n -ary gates, since they are important building blocks for combinational circuits, see e.g. the DNF circuit on slide 269.

11.3.1 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 11.3.1 (Binary Tree)** A **binary tree** is a **tree** where all **nodes** have **out-degree** 2 or 0.
- ▷ **Definition 11.3.2** A **binary tree** G is called **balanced** iff the **depth** of all **leaves** differs by at most 1, and **fully balanced**, iff the **depth** difference is 0.
- ▷ Constructing a binary tree $G_{\text{bbt}} = \langle V, E \rangle$ with n leaves
 - ▷ step 1: select some $u \in V$ as root, $(V_1 := \{u\}, E_1 := \emptyset)$
 - ▷ step 2: select $v, w \in V$ not yet in G_{bbt} and add them, $(V_i = V_{i-1} \cup \{v, w\})$
 - ▷ step 3: add two edges $\langle u, v \rangle$ and $\langle u, w \rangle$ where u is the leftmost of the shallowest nodes with $\text{outdeg}(u) = 0$, $(E_i := E_{i-1} \cup \{\langle u, v \rangle, \langle u, w \rangle\})$
 - ▷ repeat steps 2 and 3 until $i = n$ $(V = V_n, E = E_n)$
- ▷ **Example 11.3.3** 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 11.3.4** 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 $dp(G) = n > i$, 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 11.3.5** 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 11.3.4 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 11.3.3.

Depth Lemma for Balanced Trees

▷ **Lemma 11.3.6** 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 $V' := V \setminus W$, where W is the set of nodes at level $d = dp(G)$

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 11.3.7** 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 $\#(V) = 2 \cdot m - 3 + 2 = 2 \cdot m - 1$. □

□



©: Michael Kohlhase

266



In particular, the size of a binary tree is independent of its form if we fix the number of leaves. So we can optimize the depth of a binary tree by taking a balanced one without a size penalty. This will become important for building fast combinational circuits.

11.3.2 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

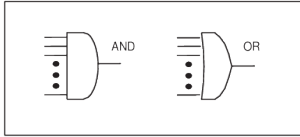
▷ **Idea:** Identify (and abbreviate) frequently occurring subgraphs

▷ **Definition 11.3.8** We define the n -ary gates by Boolean equivalence.⁷
 $AND_n(x_1, \dots, x_n) := 1 * \prod_{i=1}^n x_i$ and $OR_n(x_1, \dots, x_n) := 0 + \sum_{i=1}^n x_i$


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

▷ **Corollary 11.3.9** $C(G_n) = n - 1$ and $dp(G_n) = \lfloor \log_2(n) \rfloor$.

▷ **Notation 11.3.10**




We write the n -ary gates as



©: Michael Kohlhase

267



⁷EDNOTE: 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 11.3.11** 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

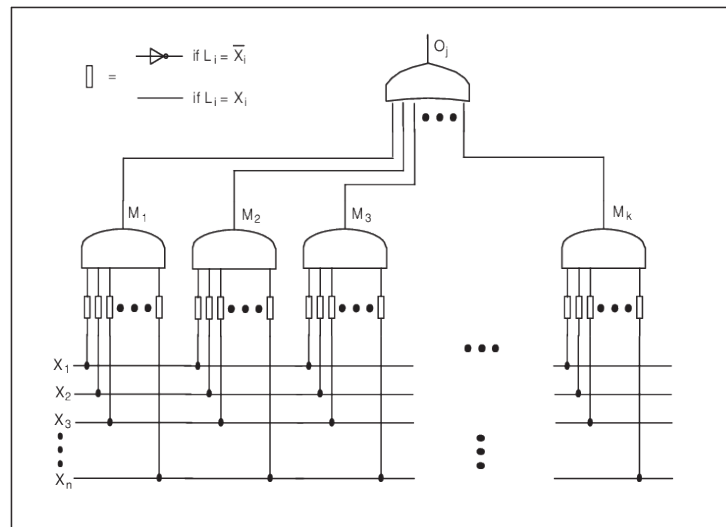
P.4 for each monomial the ANDs in the related circuit can be arranged in a balanced binary tree of depth $\lfloor \log_2(n) \rfloor$

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



Of course, the depth result is related to the first worst-case complexity result for Boolean expressions (Theorem 6.3.17); 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	and	1	0
	1	0		1	0
	0	1		0	1

- ▷ Corresponding logical connectives 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 .

SOME RIGHTS RESERVED

©: Michael Kohlhase

270

JACOBS UNIVERSITY

The significance of the NAND and NOR gates is that we can build combinational circuits 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

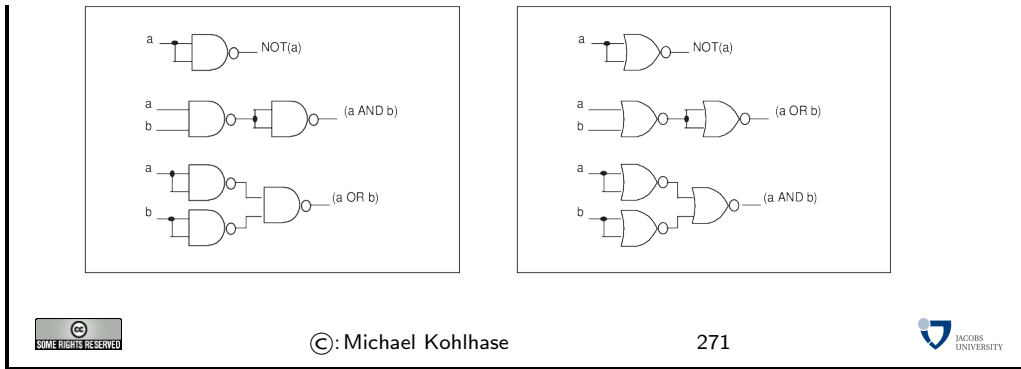
The Universality of NAND and NOR

- ▷ **Theorem 11.3.12** *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(<i>a</i>)	NAND(<i>a</i> , <i>a</i>)	NOR(<i>a</i> , <i>a</i>)
AND(<i>a</i> , <i>b</i>)	NAND(NAND(<i>a</i> , <i>b</i>), NAND(<i>a</i> , <i>b</i>))	NOR(NOR(<i>a</i> , <i>a</i>), NOR(<i>b</i> , <i>b</i>))
OR(<i>a</i> , <i>b</i>)	NAND(NAND(<i>a</i> , <i>a</i>), NAND(<i>b</i> , <i>b</i>))	NOR(NOR(<i>a</i> , <i>b</i>), NOR(<i>a</i> , <i>b</i>))

□

- ▷ 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 12

Arithmetic Circuits

12.1 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.

12.1.1 Positional Number Systems



Positional Number Systems

- ▷ **Problem:** For realistic arithmetics we need better number representations than the unary natural numbers $(|\varphi_n(\text{unary})| \in \Theta(n)$ [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 12.1.1** 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 **digit** s. $(b := \#(D_b)$ base or radix of \mathcal{N})
 - ▷ $\varphi_b: D_b \rightarrow \{\epsilon, /, \dots, /^{[b-1]}\}$ is bijective (first b unary numbers)
 - ▷ $\psi_b: D_b^+ \rightarrow \{/\}^*; \langle n_k, \dots, n_1 \rangle \mapsto \bigoplus_{i=1}^k \varphi_b(n_i) \odot \exp(/^{[b]}, /^{[i-1]})$ (extends φ_b to string code)

©: Michael Kohlhase272

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 $|\varphi_n(\text{unary})|$ is the “cost of representing n in unary representation”, we get $|\varphi_n(\text{unary})| \in \Theta(n)$. 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 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 12.1.2** 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	$FF3A12_{16}$

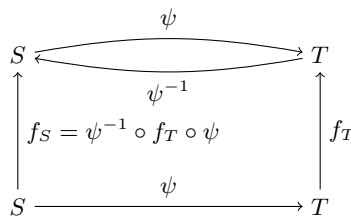
▷ **Notation 12.1.3** attach the base of \mathcal{N} to every number from \mathcal{N} . (default: decimal)

Trick: Group triples or quadruples of binary digits into recognizable chunks (add leading zeros as needed)

$$\begin{aligned}
 \triangleright \quad & \triangleright 110001101011100_2 = \underbrace{0110_2}_{6_{16}} \underbrace{0011_2}_{3_{16}} \underbrace{0101_2}_{5_{16}} \underbrace{1100_2}_{C_{16}} = 635C_{16} \\
 \triangleright \quad & \triangleright 110001101011100_2 = \underbrace{110_2}_{6_8} \underbrace{001_2}_{1_8} \underbrace{101_2}_{5_8} \underbrace{011_2}_{3_8} \underbrace{100_2}_{4_8} = 61534_8 \\
 \triangleright \quad & \triangleright FF3A_{16} = \underbrace{F_{16}}_{1111_2} \underbrace{3_{16}}_{0011_2} \underbrace{A_{16}}_{1010_2} = 111100111010_2, \quad 4721_8 = \underbrace{4_8}_{100_2} \underbrace{7_8}_{111_2} \underbrace{2_8}_{010_2} \underbrace{1_8}_{001_2} = 100111010001_2
 \end{aligned}$$

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 these number representations. We do this by using an old trick in math. If we have an operation $f_T: T \rightarrow T$ 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 $f_S: S \rightarrow S$ by $f_S(s) := \psi^{-1}(f_T(\psi(s)))$ according to the following diagram.







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 12.1.4** 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 12.1.5** Let $\mathcal{N} := \langle D_b, \varphi_b, \psi_b \rangle$ be a PNS of base b , then we define a binary function $+_b: \mathbb{N}_b \times \mathbb{N}_b \rightarrow \mathbb{N}_b$ by $(x+_by) := \psi_b^{-1}(\psi_b(x) \oplus \psi_b(y))$.
- ▷ **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)

12.1.2 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 12.1.6** 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 $a = \langle a_{n-1}, \dots, a_0 \rangle \in \mathbb{B}^n$, the notation $\langle\langle a \rangle\rangle$ is used, i.e.,

$$\langle\langle a \rangle\rangle = \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

▷ **Definition 12.1.7** An n -bit **adder** is a circuit computing the function $f_{+2}^n : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^{n+1}$ with

$$f_{+2}^n(a; b) := B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$$



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 12.1.8 A **half adder** is a circuit HA implementing the function f_{HA} in the truth table on the

▷ right.

$$f_{\text{HA}} : \mathbb{B}^2 \rightarrow \mathbb{B}^2 \quad \langle a, b \rangle \mapsto \langle c, s \rangle$$

s is called the **sum bit** and c the **carry bit**.

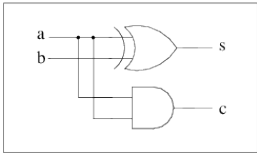
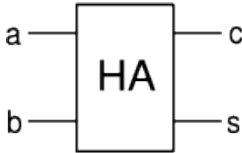
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

▷ **Note:** The carry can be computed by a simple AND, i.e., $c = \text{AND}(a, b)$, 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 $f_{HA}: \mathbb{B}^2 \rightarrow \mathbb{B}^2; \langle a, b \rangle \mapsto \langle a \oplus b, a \wedge b \rangle$
- ▷ **Note:** $f_{HA}(a, b) = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle)$, i.e., it is indeed an adder.
- ▷ We count XOR as one gate, so $C(HA) = 2$ and $dp(HA) = 1$.


©: Michael Kohlhase
277


Now that we have the half adder as a building block it is rather simple to arrive at a full adder circuit.

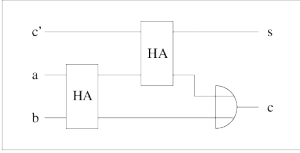
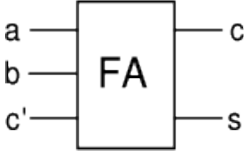
⚠, in the diagram for the full adder, and in the following, we will sometimes use a variant gate symbol for the OR gate: The symbol . It has the same outline as an AND gate, but the input lines go all the way through.



The Full Adder

- ▷ **Definition 12.1.9** The 1-bit **full adder** is a circuit FA^1 that implements the function $f_{FA}^1: \mathbb{B} \times \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}^2$ with $FA^1(a, b, c') = B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$
- ▷ The result of the full-adder is also denoted with $\langle c, s \rangle$, i.e., a carry and a sum bit. The bit c' is called the **input carry**.
- ▷ the easiest way to implement a full adder is to use two half adders and an OR gate.
- ▷ **Lemma 12.1.10 (Cost and Depth)**
 $C(FA^1) = 2C(HA) + 1 = 5$
 $dp(FA^1) = 2dp(HA) + 1 = 3$

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

and


©: Michael Kohlhase
278


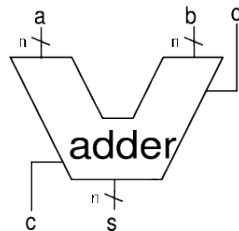
Of course adding single digits is a rather simple task, and hardly worth the effort, if this is all we can do. What we are really after, are circuits that will add n -bit binary natural numbers, so that we arrive at computer chips that can add long numbers for us.

Full n -bit Adder

▷ **Definition 12.1.11** An n -bit full adder ($n > 1$) is a circuit that corresponds to $f_{FA}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n; \langle a, b, c' \rangle \mapsto B(\langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + \langle\langle c' \rangle\rangle)$

▷ **Notation 12.1.12** 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



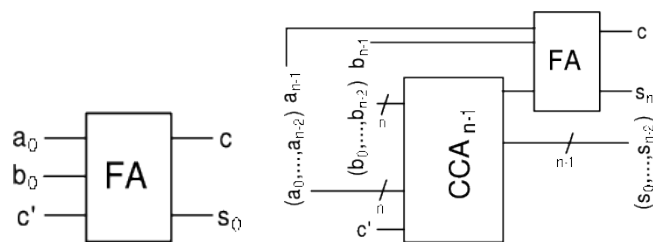
This implementation follows the intuition behind elementary school addition (only for binary numbers): we write the numbers below each other in a tabulated fashion, and from the least significant digit, we follow the process of

- 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 12.1.13** An n -bit carry chain adder CCA^n is inductively defined as


- ▷ $f_{CCA}^1(a_0, b_0, c) = FA^1(a_0, b_0, c)$
- ▷ $f_{CCA}^n(\langle a_{n-1}, \dots, a_0 \rangle, \langle b_{n-1}, \dots, b_0 \rangle, c') = \langle c, s_{n-1}, \dots, s_0 \rangle$ with
 - ▷ $\langle c, s_{n-1} \rangle = FA^{n-1}(a_{n-1}, b_{n-1}, c_{n-1})$
 - ▷ $\langle s_{n-1}, \dots, s_0 \rangle = f_{CCA}^{n-1}(\langle a_{n-2}, \dots, a_0 \rangle, \langle b_{n-2}, \dots, b_0 \rangle, c')$
- ▷ **Lemma 12.1.14 (Cost)** $C(CCA^n) \in O(n)$
- ▷ **Proof Sketch:** $C(CCA^n) = C(CCA^{n-1}) + C(FA^1) = C(CCA^{n-1}) + 5 = 5n \square$
- ▷ **Lemma 12.1.15 (Depth)** $dp(CCA^n) \in O(n)$
- ▷ **Proof Sketch:** $dp(CCA^n) \leq dp(CCA^{n-1}) + dp(FA^1) \leq dp(CCA^{n-1}) + 3 \leq 3n \square$

- ▷ 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)



©: Michael Kohlhase

281



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	?	?	?	?	5	1	3
upper sum. with carry	?	?	?	9	8					
upper sum. no carry	?	?	?	9	7					

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 12.1.16** An n -bit **conditional sum adder** CSA^n is recursively defined as
 - ▷ $f_{CSA}^n(\langle a_{n-1}, \dots, a_0 \rangle, \langle b_{n-1}, \dots, b_0 \rangle, c') = \langle c, s_{n-1}, \dots, s_0 \rangle$ where
 - ▷ $\langle c_{n/2}, s_{n/2-1}, \dots, s_0 \rangle = f_{CSA}^{n/2}(\langle a_{n/2-1}, \dots, a_0 \rangle, \langle b_{n/2-1}, \dots, b_0 \rangle, c')$
 - ▷ $\langle c, s_{n-1}, \dots, s_{n/2} \rangle = \begin{cases} f_{CSA}^{n/2}(\langle a_{n-1}, \dots, a_{n/2} \rangle, \langle b_{n-1}, \dots, b_{n/2} \rangle, 0) & \text{if } c_{n/2} = 0 \\ f_{CSA}^{n/2}(\langle a_{n-1}, \dots, a_{n/2} \rangle, \langle b_{n-1}, \dots, b_{n/2} \rangle, 1) & \text{if } c_{n/2} = 1 \end{cases}$
 - ▷ $f_{CSA}^1(a_0, b_0, c) = FA^1(a_0, b_0, c)$

SOME RIGHTS RESERVED

©: Michael Kohlhase

282

JACOBS UNIVERSITY

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 “ $(C \text{ and } A) \text{ or } (\text{not } C \text{ and } B)$ ”.

The Multiplexer

- ▷ **Definition 12.1.17** An n -bit **multiplexer** MUX^n is a circuit which implements the function $f_{MUX}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B}^n$ with

$$f_{MUX}^n(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, s) = \begin{cases} \langle a_{n-1}, \dots, a_0 \rangle & \text{if } s = 0 \\ \langle b_{n-1}, \dots, b_0 \rangle & \text{if } s = 1 \end{cases}$$
- ▷ **Idea:** A multiplexer chooses between two n -bit input vectors A and B depend-

ing on the value of the control bit s .

▷ Cost and depth: $C(\text{MUX}^n) = 3n + 1$ and $\text{dp}(\text{MUX}^n) = 3$.

©: Michael Kohlhase 283

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

▷ $\text{dp}(\text{CSA}^n) \leq \text{dp}(\text{CSA}^{n/2}) + \text{dp}(\text{MUX}^{n/2+1})$

▷ solve the recursive equation:

$$\begin{aligned} \text{dp}(\text{CSA}^n) &\leq \text{dp}(\text{CSA}^{n/2}) + \text{dp}(\text{MUX}^{n/2+1}) \\ &\leq \text{dp}(\text{CSA}^{n/2}) + 3 \\ &\leq \text{dp}(\text{CSA}^{n/4}) + 3 + 3 \\ &\leq \text{dp}(\text{CSA}^{n/8}) + 3 + 3 + 3 \\ &\dots \\ &\leq \text{dp}(\text{CSA}^{n^{2^{-i}}}) + 3i \\ &\leq \text{dp}(\text{CSA}^1) + 3\log_2(n) \\ &\leq 3\log_2(n) + 3 \end{aligned}$$

©: Michael Kohlhase 284

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

▷ $C(\text{CSA}^n) = 3C(\text{CSA}^{n/2}) + C(\text{MUX}^{n/2+1})$.

▷ 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 12.1.18 (Master's Theorem for Recursive Equations)** Given the recursively defined function $f: \mathbb{N} \rightarrow \mathbb{R}$, 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 $g: \mathbb{N} \rightarrow \mathbb{R}$, then $f(b^k) = ca^k + \sum_{i=0}^{k-1} a^i g(b^{k-i})$

▷ We have

$$\begin{aligned} C(\text{CSA}^n) &= 3C(\text{CSA}^{n/2}) + C(\text{MUX}^{n/2+1}) \\ &= 3C(\text{CSA}^{n/2}) + 3(n/2 + 1) + 1 \\ &= 3C(\text{CSA}^{n/2}) + \frac{3}{2}n + 4 \end{aligned}$$

▷ So, $C(\text{CSA}^n)$ 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 $c = C(f_{\text{CSA}}^1) = C(\text{FA}^1) = 5$

▷ thus $C(\text{CSA}^n) = 5 \cdot 3^{\log_2(n)} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4$

▷ **Note:** $a^{\log_2(n)} = 2^{\log_2(a) \log_2(n)} = 2^{\log_2(a) \cdot \log_2(n)} = 2^{\log_2(n) \log_2(a)} = n^{\log_2(a)}$

$$\begin{aligned} C(\text{CSA}^n) &= 5 \cdot 3^{\log_2(n)} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \frac{3}{2}n \cdot 2^{-i} + 4 \\ &= 5n^{\log_2(3)} + \sum_{i=1}^{\log_2(n)} n \frac{3^i}{2} + 4 \\ &= 5n^{\log_2(3)} + n \cdot \sum_{i=1}^{\log_2(n)} \frac{3^i}{2} + 4\log_2(n) \\ &= 5n^{\log_2(3)} + 2n \cdot \frac{3^{\log_2(n)+1}}{2} - 1 + 4\log_2(n) \\ &= 5n^{\log_2(3)} + 3n \cdot n^{\log_2(\frac{3}{2})} - 2n + 4\log_2(n) \\ &= 8n^{\log_2(3)} - 2n + 4\log_2(n) \in O(n^{\log_2(3)}) \end{aligned}$$

▷ **Compare with:** $C(\text{CCA}^n) \in O(n)$ $\text{dp}(\text{CCA}^n) \in O(n)$

▷ 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.

12.2 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 12.2.1** ($(n + 1)$ -bit signed binary number system)

$$\langle\langle a_n, \dots, a_0 \rangle\rangle^- := \begin{cases} \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle & \text{if } a_n = 0 \\ -\langle\langle a_{n-1}, \dots, a_0 \rangle\rangle & \text{if } a_n = 1 \end{cases}$$

- ▷ **Note:** We need to fix string length to identify the sign bit. (leading zeroes)
- ▷ **Example 12.2.2** 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 $\#\{\langle\langle s \rangle\rangle^- \mid s \in \mathbb{B}^n\} = 2 \cdot (2^{n-1}) + 1 = 2^n - 1$ numbers all in all

▷ One number must be represented twice (But there are 2^n strings of length n .)

▷ $10\dots 0$ and $00\dots 0$ both represent the zero as $-1 \cdot 0 = 1 \cdot 0$.

▷ We could build arithmetic circuits using this, but there is a more elegant way!

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

©: Michael Kohlhase
287

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 12.2.3** Given the binary string $a = \langle a_n, \dots, a_0 \rangle \in \mathbb{B}^{n+1}$, 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

$$\begin{aligned}
 \langle\langle a \rangle\rangle_n^{2s} &= -a_n \cdot 2^n + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\
 &= -a_n \cdot 2^n + \sum_{i=0}^{n-1} a_i \cdot 2^i
 \end{aligned}$$

▷ **Notation 12.2.4** 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

©: Michael Kohlhase
288

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 $b = \langle b_n, \dots, b_0 \rangle$ be a number in the $n + 1$ -bit two's complement system, then

- ▷ Positive numbers and the zero have a sign bit 0, i.e., $b_n = 0 \Leftrightarrow (\langle\langle b \rangle\rangle_n^{2s} \geq 0)$.
- ▷ Negative numbers have a sign bit 1, i.e., $b_n = 1 \Leftrightarrow \langle\langle b \rangle\rangle_n^{2s} < 0$.
- ▷ For positive numbers, the two's complement representation corresponds to the normal binary number representation, i.e., $b_n = 0 \Leftrightarrow \langle\langle b \rangle\rangle_n^{2s} = \langle\langle b \rangle\rangle$
- ▷ There is a unique representation of the number zero in the n -bit two's complement system, namely $B_n^{2s}(0) = \langle 0, \dots, 0 \rangle$.
- ▷ This number system has an asymmetric range $\mathcal{R}^{2s}_n := \{-2^n, \dots, 2^n - 1\}$.



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 its negative sibling) can be achieved by two very simple operations: flipping all the zeros and ones, and incrementing.

The Structure Theorem for TCN

- ▷ **Theorem 12.2.5** Let $a \in \mathbb{B}^{n+1}$ be a binary string, then $-\langle\langle a \rangle\rangle_n^{2s} = \langle\langle \bar{a} \rangle\rangle_n^{2s} + 1$, where \bar{a} is the pointwise bit complement of a .

- ▷ **Proof Sketch:** By calculation using the definitions:

$$\begin{aligned}
 \langle\langle \bar{a}_n, \bar{a}_{n-1}, \dots, \bar{a}_0 \rangle\rangle_n^{2s} &= -\bar{a}_n \cdot 2^n + \langle\langle \bar{a}_{n-1}, \dots, \bar{a}_0 \rangle\rangle \\
 &= \bar{a}_n \cdot -2^n + \sum_{i=0}^{n-1} \bar{a}_i \cdot 2^i \\
 &= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 1 - a_i \cdot 2^i \\
 &= 1 - a_n \cdot -2^n + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i \cdot 2^i \\
 &= -2^n + a_n \cdot 2^n + 2^{n-1} - \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\
 &= (-2^n + 2^n) + a_n \cdot 2^n - \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle - 1 \\
 &= -(a_n \cdot -2^n + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle) - 1 \\
 &= -\langle\langle a \rangle\rangle_n^{2s} - 1
 \end{aligned}$$

□



A first simple application of the TCN structure theorem is that we can use our existing conversion routines (for binary natural numbers) to do TCN conversion (for integers).

Application: Converting from and to TCN?

- ▷ to convert an integer $-z \in \mathbb{Z}$ with $z \in \mathbb{N}$ into an n -bit TCN
 - ▷ generate the n -bit binary number representation $B(z) = \langle b_{n-1}, \dots, b_0 \rangle$
 - ▷ complement it to $\overline{B(z)}$, i.e., the bitwise negation \bar{b}_i of $B(z)$

- ▷ increment (add 1) $\overline{B(z)}$, i.e. compute $B(\langle\overline{B(z)}\rangle + 1)$
- ▷ to convert a negative n -bit TCN $b = \langle b_{n-1}, \dots, b_0 \rangle$, into an integer
 - ▷ decrement b , (compute $B(\langle b \rangle - 1)$)
 - ▷ complement it to $\overline{B(\langle b \rangle - 1)}$
 - ▷ compute the decimal representation and negate it to $-\langle\overline{B(\langle b \rangle - 1)}\rangle$



Subtraction and Two's Complement Numbers

- ▷ **Idea:** With negative numbers use our adders directly
- ▷ **Definition 12.2.6** An n -bit **subtractor** is a circuit that implements the function $f_{\text{SUB}}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n$ such that

$$f_{\text{SUB}}^n(a, b, b') = B_n^{2s}(\langle a \rangle_n^{2s} - \langle b \rangle_n^{2s} - b')$$

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 $\langle a \rangle_n^{2s} - \langle b \rangle_n^{2s} = \langle a \rangle_n^{2s} + (-\langle b \rangle_n^{2s}) = \langle a \rangle_n^{2s} + \langle \bar{b} \rangle_n^{2s} + 1$
- ▷ **Idea:** Can we implement an n -bit subtracter as $f_{\text{SUB}}^n(a, b, b') = \text{FA}^n(a, \bar{b}, \bar{b}')$?
- ▷ **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 12.2.7** An n -bit **two's complement adder** ($n > 1$) is a circuit that corresponds to the function $f_{\text{TCA}}^n: \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}^n$, such that $f_{\text{TCA}}^n(a, b, c') = B_n^{2s}(\langle a \rangle_n^{2s} + \langle b \rangle_n^{2s} + c')$ for all $a, b \in \mathbb{B}^n$ and $c' \in \mathbb{B}$.
- ▷ **Theorem 12.2.8** $f_{\text{TCA}}^n = f_{\text{FA}}^n$ (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 $\text{TCA} \text{circ} \text{FN} n$ and the full adder function f_{FA}^n 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 12.2.9** Let $a = \langle a_n, \dots, a_0 \rangle \in \mathbb{B}^{n+1}$ be a binary string, then $\langle\langle a_n, \dots, a_0 \rangle\rangle_{n+1}^{2s} = \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle_n^{2s}$.

▷ **Proof Sketch:** By calculation:


$$\begin{aligned} \langle\langle a_n, \dots, a_0 \rangle\rangle_{n+1}^{2s} &= -a_n \cdot 2^{n+1} + \langle\langle a_n, \dots, a_0 \rangle\rangle \\ &= -a_n \cdot 2^{n+1} + a_n \cdot 2^n + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\ &= a_n \cdot (-2^{n+1} + 2^n) + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\ &= a_n \cdot (-2 \cdot 2^n + 2^n) + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\ &= -a_n \cdot 2^n + \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle \\ &= \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle_n^{2s} \end{aligned}$$

□



©: Michael Kohlhase

294



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 12.2.10** Let $a, b \in \mathbb{B}^{n+1}$ and $c \in \mathbb{B}$ with $a = \langle a_n, \dots, a_0 \rangle$ and $b = \langle b_n, \dots, b_0 \rangle$, then we call $(ic_k(a, b, c))$, the k -th intermediate carry of a , b , and c , iff

$$\langle\langle ic_k(a, b, c), s_{k-1}, \dots, s_0 \rangle\rangle = \langle\langle a_{k-1}, \dots, a_0 \rangle\rangle + \langle\langle b_{k-1}, \dots, b_0 \rangle\rangle + c$$

for some $s_i \in \mathbb{B}$.


▷ **Theorem 12.2.11** Let $a, b \in \mathbb{B}^n$ and $c \in \mathbb{B}$, then

- 1) $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \in \mathcal{R}^{2s}_n$, 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 $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = \langle\langle s \rangle\rangle_n^{2s}$, where $\langle\langle ic_{n+1}(a, b, c), s_n, \dots, s_0 \rangle\rangle = \langle\langle a \rangle\rangle + \langle\langle b \rangle\rangle + c$.



©: Michael Kohlhase

295



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

$a' = \langle a_{n-1}, \dots, a_0 \rangle$ and $b' = \langle b_{n-1}, \dots, b_0 \rangle$.

P.1 Then

$$\begin{aligned} \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c &= \langle\langle a_{n-1}, \dots, a_0 \rangle\rangle + \langle\langle b_{n-1}, \dots, b_0 \rangle\rangle + c \\ &= \langle\langle \text{ic}_n(a, b, c), s_{n-1}, \dots, s_0 \rangle\rangle \end{aligned}$$

$$\text{and } a_n + b_n + (\text{ic}_n(a, b, c)) = \langle\langle \text{ic}_{n+1}(a, b, c), s_n \rangle\rangle.$$

P.2 We have to consider three cases

P.2.1 $a_n = b_n = 0$:

P.2.1.1 $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ are both positive, so $(\text{ic}_{n+1}(a, b, c)) = 0$ and furthermore

$$\begin{aligned} (\text{ic}_n(a, b, c)) = 0 &\Leftrightarrow \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c \leq 2^n - 1 \end{aligned}$$

P.2.1.2 Hence,

$$\begin{aligned} \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= \langle\langle s_{n-1}, \dots, s_0 \rangle\rangle \\ &= \langle\langle 0, s_{n-1}, \dots, s_0 \rangle\rangle = \langle\langle s \rangle\rangle_n^{2s} \end{aligned}$$

□

P.2.2 $a_n = b_n = 1$:

P.2.2.1 $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ are both negative, so $(\text{ic}_{n+1}(a, b, c)) = 1$ and furthermore $(\text{ic}_n(a, b, c)) = 1$, iff $\langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq 2^n$, which is the case, iff $\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \geq -2^n$

P.2.2.2 Hence,

$$\begin{aligned} \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a' \rangle\rangle + -2^n + \langle\langle b' \rangle\rangle + c \\ &= -2^{n+1} + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\ &= -2^{n+1} + \langle\langle 1, s_{n-1}, \dots, s_0 \rangle\rangle \\ &= -2^n + \langle\langle s_{n-1}, \dots, s_0 \rangle\rangle \\ &= \langle\langle s \rangle\rangle_n^{2s} \end{aligned}$$

□

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 $(\text{ic}_{n+1}(a, b, c)) = (\text{ic}_n(a, b, c))$)

P.2.3.2 Hence, the sum of $\langle\langle a \rangle\rangle_n^{2s}$ and $\langle\langle b \rangle\rangle_n^{2s}$ is in the admissible range \mathcal{R}^{2s}_n as

$$\langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c = \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c - 2^n$$

$$\text{and } 0 \leq \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \leq 2^{n+1} - 1$$

P.2.3.3 So we have

$$\begin{aligned}
 \langle\langle a \rangle\rangle_n^{2s} + \langle\langle b \rangle\rangle_n^{2s} + c &= -2^n + \langle\langle a' \rangle\rangle + \langle\langle b' \rangle\rangle + c \\
 &= -2^n + \langle\langle \text{ic}_n(a, b, c), s_{n-1}, \dots, s_0 \rangle\rangle \\
 &= -(1 - (\text{ic}_n(a, b, c))) \cdot 2^n + \langle\langle s_{n-1}, \dots, s_0 \rangle\rangle \\
 &= \langle\langle \overline{\text{ic}_n(a, b, c)}, s_{n-1}, \dots, s_0 \rangle\rangle_n^{2s}
 \end{aligned}$$

P.2.3.4 Furthermore, we can conclude that $\langle\langle \overline{\text{ic}_n(a, b, c)}, s_{n-1}, \dots, s_0 \rangle\rangle_n^{2s} = \langle\langle s \rangle\rangle_n^{2s}$ as $s_n = a_n \oplus b_n \oplus (\text{ic}_n(a, b, c)) = 1 \oplus (\text{ic}_n(a, b, c)) = \overline{\text{ic}_n(a, b, c)}$. \square

P.3 Thus we have considered all the cases and completed the proof. \square



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.



12.3 Towards an Algorithmic-Logic Unit

The most important application of the main TCN theorem is that we can build a combinational circuit that can add and subtract (depending on a control bit). This is actually the first instance of a concrete programmable computation device we have seen up to date (we interpret the control bit as a program, which changes the behavior of the device). The fact that this is so simple, it only runs two programs should not deter us; we will come up with more complex things later.

Building an Add/Subtract Unit

▷ **Idea:** Build a Combinational Circuit that can add and subtract (sub = 1 \rightsquigarrow subtract)

▷ If sub = 0, then the circuit acts like an adder ($a \oplus 0 = a$)

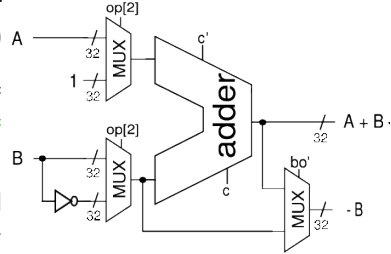
▷ If sub = 1, let $S := \langle\langle a \rangle\rangle_n^{2s} + \langle\langle \overline{b_{n-1}}, \dots, \overline{b_0} \rangle\rangle_n^{2s} + 1$ ($a \oplus 0 = 1 - a$)

▷ For $s \in \mathcal{R}^{2s}_n$ the TCN main theorem and the TCN structure theorem together guarantee

$$\begin{aligned} s &= \langle\langle a \rangle\rangle_n^{2s} + \langle\langle \overline{b_{n-1}}, \dots, \overline{b_0} \rangle\rangle_n^{2s} + 1 \\ &= \langle\langle a \rangle\rangle_n^{2s} - \langle\langle b \rangle\rangle_n^{2s} - 1 + 1 \end{aligned}$$

▷ **Summary:** We have built a combinational circuit that can perform 2 arithmetic operations depending on a control bit.

▷ **Idea:** Extend this to a **arithmetic logic unit (ALU)** with more operations(+, -, *, /, n-AND, n-OR,...)



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 13

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 modeled as acyclic labeled 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.

13.1 Sequential Logic Circuits

Sequential Logic Circuits

- ▷ In combinational circuits, outputs only depend on inputs (no state)
- ▷ We have disregarded all timing issues (except for favoring shallow circuits)
- ▷ **Definition 13.1.1** Circuits that remember their current output or state are often called sequential logic circuits.
- ▷ **Example 13.1.2** A counter, where the next number to be output is determined by the current number stored.
- ▷ Sequential logic circuits need some ability to store the current state



©: Michael Kohlhase

299

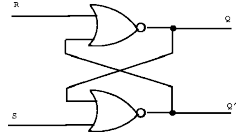


Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits. We will investigate basic circuits that have the ability to store bits of data. We will start with the simplest possible memory element, and develop more elaborate 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 13.1.3** 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.



©: Michael Kohlhase

300



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*.

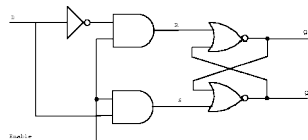
$S = 0$ **and** $R = 0$ Assume the flipflop is set ($Q = 1$ and $Q' = 0$), then the output of the top NOR gate remains at $Q = 1$ and the bottom NOR gate stays at $Q' = 0$. Similarly, when the flipflop is in a reset state ($Q = 0$ and $Q' = 1$), it will remain there with this input combination. Therefore, 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 13.1.4** 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 input** s.

- ▷ 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.”



©:Michael Kohlhase

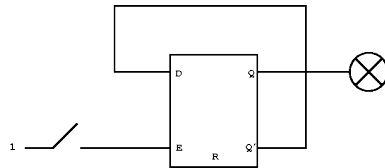
301



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)



©:Michael Kohlhase

302



In the on/off circuit, the external inputs (buttons) were connected to the E input.

Definition 13.1.5 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.

13.2 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.

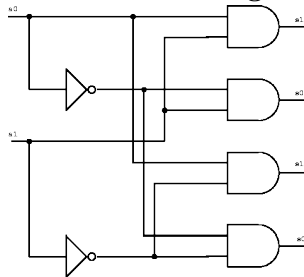
- ▷ We will consider the following questions:
 - ▷ What is the physical structure of RAM chips?
 - ▷ How are addresses used to select a particular storage element?
 - ▷ What do individual storage elements look like?
 - ▷ How is reading and writing distinguished?



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 13.2.1** The n -bit **address decoder** ADL^n has a n inputs and 2^n outputs. $f_{ADL}^m(a) = \langle b_1, \dots, b_{2^n} \rangle$, where $b_i = 1$, iff $i = \langle\langle a \rangle\rangle$.
- ▷ **Example 13.2.2 (Address decoder logic for 2-bit addresses)**

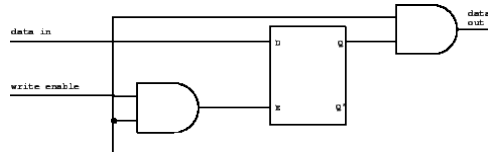


Now we can combine an n -bit address decoder as sketched by the example above, with n **D-flipflops** to get a RAM element.

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 13.2.3** A Storage Element is given by the following diagram



©: Michael Kohlhase

305



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.



©: Michael Kohlhase

306



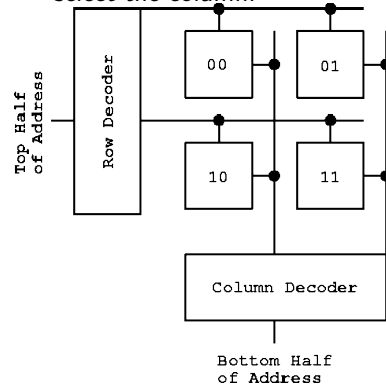
One aspect of this is particularly interesting – and user-visible in the sense that the division of storage addresses is divided into a high- and low part of the address. So we we will briefly discuss it here.

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.

13.3 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 13.3.1** A **bit** (a contraction of “binary digit”) is the basic unit of capacity of a data storage device or 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 13.3.2** 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 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 **balanced**s).

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 13.3.3 (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 13.3.4** 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)	<i>binary digit 0/1</i>
Byte (B)	<i>8 bit</i>
2 Bytes	A Unicode character in UTF.
10 Bytes	your name.
Kilobyte (kB)	<i>1,000 bytes OR 10^3 bytes</i>
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (MB)	<i>1,000,000 bytes OR 10^6 bytes</i>
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)	<i>1,000,000,000 bytes or 10^9 bytes</i>
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)	<i>1,000,000,000,000 bytes or 10^{12} bytes</i>
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)	<i>1,000,000,000,000,000 bytes or 10^{15} bytes</i>
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)	<i>1,000,000,000,000,000,000 bytes or 10^{18} bytes</i>
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)	<i>1,000,000,000,000,000,000,000 bytes or 10^{21} bytes</i>
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 14

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 $\mathcal{L}(\text{VM})$ and write a $\mathcal{L}(\text{VM})$ -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 $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ 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].



14.1 How to Build and Program a Computer (in Principle)

In this section, we will combine the arithmetic/logical units from Chapter 11 with the storage elements (RAM) from Section 13.1 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 **register** s: few (named) n -bit memory cells near the ALU

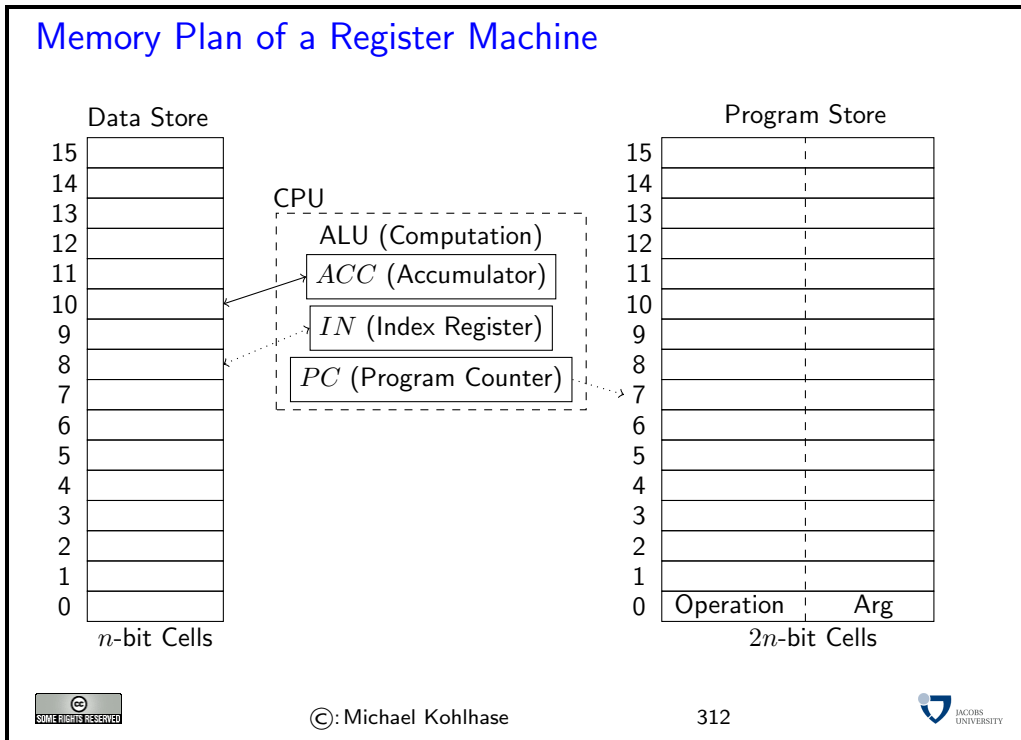
- ▷ **program counter** (*PC*) (points to current command in program store)
- ▷ **accumulator** (*ACC*) (the *a* input and output of the ALU)
- ▷ add **RAM**: lots of **random access memory** (elsewhere)
- ▷ **program store**: $2n$ -bit memory cells (addressed by $P: \mathbb{N} \rightarrow \mathbb{B}^{2n}$)
- ▷ **data store**: n -bit memory cells (word s addressed by $D: \mathbb{N} \rightarrow \mathbb{B}^n$)
- ▷ add a **memory management unit** (**MMU**) (move values between RAM and registers)
- ▷ program it in **assembler language** (lowest level of programming)


©: Michael Kohlhase
311


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.

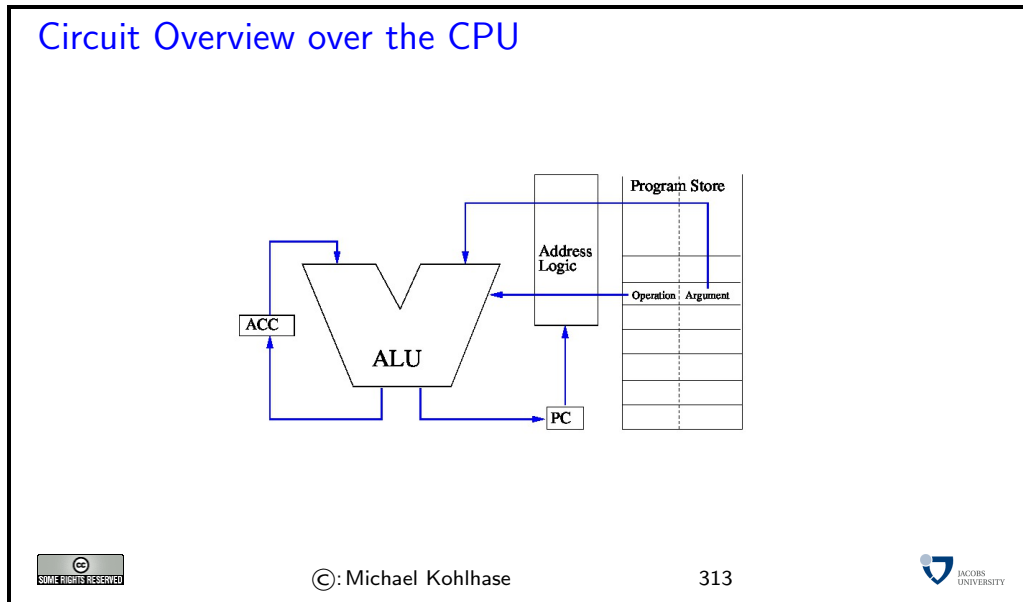
The program store is a sequence of up to 2^n memory $2n$ -bit memory cells, which can be accessed (written to and queried) randomly i.e. by referencing their position in the sequence; we do not have to access them by some fixed regime, e.g. one after the other, in sequence (hence the name random access memory: RAM). We address the Program store by a function $P: \mathbb{N} \rightarrow \mathbb{B}^{2n}$. The data store is also RAM, but a sequence or n -bit cells, which is addressed by the function $D: \mathbb{N} \rightarrow \mathbb{B}^n$.

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.



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.



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 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 if 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 14.1.1 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 14.1.2 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 14.1.3 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 14.1.4** 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 14.1.5** 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)

 ©: Michael Kohlhase 315 

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 14.1.6** Introduce an **index register** IN and register instructions

instruction	effect	PC	comment
<code>RLOAD i</code>	$ACC := D(IN)$	$PC := PC + 1$	relative load
<code>RSTORE i</code>	$D(IN) := ACC$	$PC := PC + 1$	relative store
<code>MVAI i</code>	$IN := ACC$	$PC := PC + 1$	move value
<code>MVIA i</code>	$ACC := IN$	$PC := PC + 1$	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	$D(x) = D(IN) := ACC = y$

▷

 ©: Michael Kohlhase 316 

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 314, 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 14.1.7

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 14.1.8 (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 14.1.9** Let $D(0) = n$, $D(1) = a$, and $D(2) = b$, copy the values of cells $a, \dots, a+n-1$ to cells $b, \dots, b+n-1$, 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 14.1.10** 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 14.1.11** The induction hypotheses are called **loop invariants**.



14.2 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 14.2.1 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 14.2.2** An `ASM` program `VM` is called a **virtual machine** for $\mathcal{L}(\text{VM})$, iff `VM` inputs a $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$. (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.

14.2.1 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 14.2.3** 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 14.2.4** The $\mathcal{L}(\text{VM})$ program “con 4 con 7 add” pushes $7 + 4 = 11$ to the stack.

▷ **Example 14.2.5** 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 · 2
▷	con 3 con 4 mul	4 · 3
	sub	4 · 3 - 7 · 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 14.2.6** 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 14.2.7** 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

<pre> con 2 con 1 leq cjp 9 con 3 con 4 sub jp 7 con 5 con 7 mul halt </pre>	<pre> if 1 ≤ 2 then 4 - 3 else 7 · 5 </pre>
	©: Michael Kohlhase
321	

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 14.2.8 Imperative access to variables:** Let $\mathcal{S}(i)$ be the number at stack position i .

instruction	effect	VPC
<code>peek i</code>	<code>push $\mathcal{S}(i)$</code>	<code>VPC := VPC + 2</code>
<code>poke i</code>	<code>pop x $\mathcal{S}(i) := x$</code>	<code>VPC := VPC + 2</code>

▷ **Example 14.2.9** The program “`con 5 con 7 peek 0 peek 1 add poke 1 mul halt`” computes $5 \cdot (7 + 5) = 60$.


©: Michael Kohlhase
322


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 $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ 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 14.2.10** Consider the following program that computes 12! and the corresponding $\mathcal{L}(\text{VM})$ program:

<pre> var n := 12; var a := 1; while 2 <= n do (a := a * n; n := n - 1;) return a; </pre>	<pre> 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 </pre>
--	--

▷ 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 poking the stack position (must remember that)

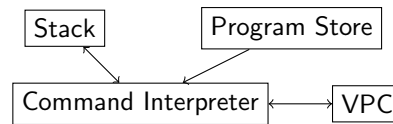


We see that again, only the result of the computation is left on the stack. In fact, the code snippet consists of two variable declarations (which extend the stack) and one `while` statement, which does not, and the `return` statement, which extends the stack again. In this case, we see that even though the `while` statement does not extend the stack it does change the stack below by the variable assignments (implemented as `poke` in $\mathcal{L}(\text{VM})$). We will use the example above as guiding intuition for a compiler from a simple imperative language to $\mathcal{L}(\text{VM})$ byte code below. But first we build a virtual machine for $\mathcal{L}(\text{VM})$.

14.2.2 Building a Virtual Machine

A Virtual Machine for Functional Programming

- ▷ We will build a stack-based virtual machine; this will have four components



- ▷ The **stack** is a memory segment operated as a “last-in-first-out” LIFO sequence
- ▷ The **program store** is a memory segment interpreted as a sequence of instructions
- ▷ The **command interpreter** is a ASM program that interprets commands from the program store and operates on the stack.
- ▷ The **virtual program counter (VPC)** is a register that acts as the pointer to the current instruction in the program store.
- ▷ The virtual machine starts with the empty stack and VPC at the beginning of the program.



We will now build a virtual machine for $\mathcal{L}(\text{VM})$ along the specification above.

To make our implementation of the virtual machine more convenient, we will extend `ASM` with a couple of convenience features. Note that these features do not extend the theoretical expressivity of `ASM` (i.e. they do not extend the range of programs that `ASM`), since all new commands can be replaced by regular language constructs.

Extending `REMA` and `ASM`

- ▷ Give ourselves two more index registers: $IN_1 := IN$, IN_2 , and IN_3 and

instruction	effect	PC	comment
LOADIN $j\ i$	$ACC := D(IN_j + i)$	$PC := PC + 1$	relative load
STOREIN $j\ i$	$D(IN_j + i) := ACC$	$PC := PC + 1$	relative store
MOVE $S\ T$	$T := S$	$PC := PC + 1$	move value from reg. S to reg. T

We will use a syntactic variant of ASM for transparency

- ▷ ▷ JUMP and JUMP= with labels of the form $\langle foo \rangle$ (compute relative jump distances automatically)
- ▷ inc R for MOVE $R\ ACC$, ADDI 1, MOVE $ACC\ R$ (dec R similar)
- ▷ note that inc R and dec R overwrite the current ACC (take care of it)
- ▷ All additions can be eliminated by substitution.



Note that the LOADIN are not binary instructions, but that this is just a short notation for unary instructions LOADIN 1 and LOADIN 2 (and similarly for MOVE $S\ T$).

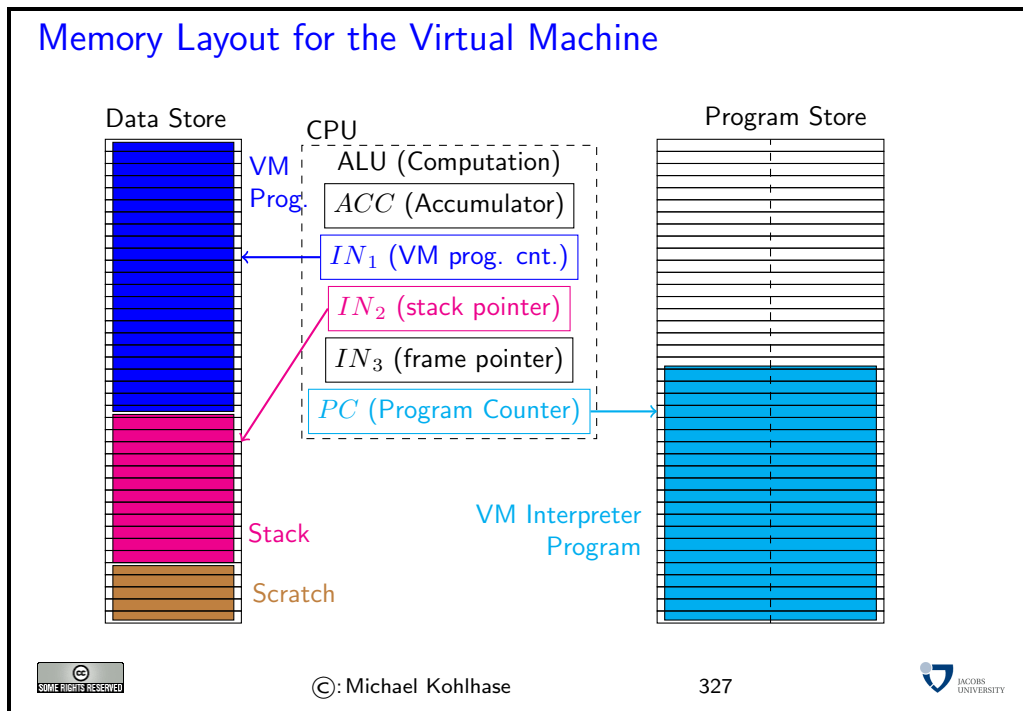
Note furthermore that the addition logic in LOADIN j is simply for convenience (most assembler languages have it, since working with address offsets is commonplace). We could have always imitated this by a simpler relative load command and an ADD instruction.

A Virtual Machine for $\mathcal{L}(\text{VM})$

- ▷ We need to build a concrete ASM program that acts as a virtual machine for $\mathcal{L}(\text{VM})$.
- ▷ Choose a concrete register machine size: e.g. 32-bit words (like in a PC)
- ▷ Choose memory layout in the data store
 - ▷ the VM stack: $D(8)$ to $D(2^{24} - 1)$, and (need the first 8 cells for VM data)
 - ▷ the $\mathcal{L}(\text{VM})$ program store: $D(2^{24})$ to $D(2^{32} - 1)$
 - ▷ We represent the virtual program counter VPC by the index register IN_1 and the stack pointer by the index register IN_2 (with offset 8).
 - ▷ We will use $D(0)$ as an argument store.
- ▷ choose a numerical representation for the $\mathcal{L}(\text{VM})$ instructions: (have lots of space)
 - halt $\mapsto 0$, add $\mapsto 1$, sub $\mapsto 2$, ...



Recall that the virtual machine VM is a ASM program, so it will reside in the REMA program store. This is the program executed by the register machine. So both the VM stack and the $\mathcal{L}(\text{VM})$ program have to be stored in the REMA data store (therefore we treat $\mathcal{L}(\text{VM})$ programs as sequences of words and have to do counting acrobatics for instructions of differing length). We somewhat arbitrarily fix a boundary in the data store of REMA at cell number $2^{24} - 1$. We will also need a little piece of scratch-pad memory, which we locate at cells 0-7 for convenience (then we can simply address with absolute numbers as addresses).



With these extensions, it is quite simple to write the ASM code that implements the virtual machine VM.

The first part of VM is a simple jump table, a piece of code that does nothing else than distributing the program flow according to the (numerical) instruction head. We assume that this program segment is located at the beginning of the program store, so that the REMA program counter points to the first instruction. This initializes the VM program counter and its stack pointer to the first cells of their memory segments. We assume that the $\mathcal{L}(\text{VM})$ program is already loaded in its proper location, since we have not discussed input and output for REMA.

Starting VM: the Jump Table

label	instruction	effect	comment
$\langle jt \rangle$	LOADI 2^{24}	$ACC := 2^{24}$	load VM start address
	MOVE $ACC\ IN_1$	$VPC := ACC$	set VPC
	LOADI 7	$ACC := 7$	load top of stack address
	MOVE $ACC\ IN_2$	$SP := ACC$	set SP
	LOADIN 1 0	$ACC := D(IN_1)$	load instruction
	JUMP = $\langle halt \rangle$		goto $\langle halt \rangle$
	ADDI -1		next instruction code
	JUMP = $\langle add \rangle$		goto $\langle add \rangle$
$\langle halt \rangle$	ADDI -1		next instruction code
	JUMP = $\langle sub \rangle$		goto $\langle sub \rangle$
	STOP 0		stop
	⋮	⋮	⋮
	⋮	⋮	⋮

©: Michael Kohlhase 328 JACOBS UNIVERSITY

Now it only remains to present the ASM programs for the individual $\mathcal{L}(\text{VM})$ instructions. We will start with the arithmetical operations.

The code for `con` is absolutely straightforward: we increment the VM program counter to point to the argument, read it, and store it to the cell the (suitably incremented) VM stack pointer points to. Once procedure has been executed we increment the VM program counter again, so that it points to the next $\mathcal{L}(\text{VM})$ instruction, and jump back to the beginning of the jump table.



For the `add` instruction we have to use the scratch pad area, since we have to pop two values from the stack (and we can only keep one in the accumulator). We just cache the first value in cell 0 of the program store.

Implementing Arithmetic Operators

label	instruction	effect	comment
$\langle \text{con} \rangle$	<code>inc IN_1</code>	$VPC := VPC + 1$	point to arg
	<code>inc IN_2</code>	$SP := SP + 1$	prepare push
	<code>LOADIN 1 0</code>	$ACC := D(VPC)$	read arg
	<code>STOREIN 2 0</code>	$D(SP) := ACC$	store for push
	<code>inc IN_1</code>	$VPC := VPC + 1$	point to next
	<code>JUMP $\langle jt \rangle$</code>		jump back
$\langle \text{add} \rangle$	<code>LOADIN 2 0</code>	$ACC := D(SP)$	read arg 1
	<code>STORE 0</code>	$D(0) := ACC$	cache it
	<code>dec IN_2</code>	$SP := SP - 1$	pop
	<code>LOADIN 2 0</code>	$ACC := D(SP)$	read arg 2
	<code>ADD 0</code>	$ACC := ACC + D(0)$	add cached arg 1
	<code>STOREIN 2 0</code>	$D(SP) := ACC$	store it
	<code>inc IN_1</code>	$VPC := VPC + 1$	point to next
	<code>JUMP $\langle jt \rangle$</code>		jump back

▷ `sub`, similar to `add`.

▷ `mul`, and `leq` need some work.


©: Michael Kohlhase
329


We will not go into detail for the other arithmetic commands, for example, `mul` could be implemented as follows:

label	instruction	effect	comment
$\langle \text{mul} \rangle$	<code>dec IN_2</code>	$SP := SP - 1$	
	<code>LOADI 0</code>		
	<code>STORE 1</code>	$D(1) := 0$	initialize result
$\langle \text{loop} \rangle$	<code>LOADIN 2 1</code>	$ACC := D(SP + 1)$	read arg 1
	<code>STORE 0</code>	$D(0) := ACC$	initialize counter to arg 1
	<code>JUMP= $\langle \text{end} \rangle$</code>		if counter=0, we are finished
	<code>LOADIN 2 0</code>	$ACC := D(SP)$	read arg 2
$\langle \text{loop} \rangle$	<code>ADD 1</code>	$ACC := ACC + D(1)$	current sum increased by arg 2
	<code>STORE 1</code>	$D(1) := ACC$	cache result
$\langle \text{end} \rangle$	<code>LOAD 0</code>		
	<code>ADDI -1</code>		
	<code>STORE 0</code>	$D(0) := D(0) - 1$	decrease counter by 1
	<code>JUMP loop</code>		repeat addition
	<code>LOAD 1</code>		load result
$\langle \text{end} \rangle$	<code>STOREIN 2 0</code>		push it on stack
	<code>inc IN_1</code>		
	<code>JUMP $\langle jt \rangle$</code>		back to jump table

Note that `mul` and `leq` are the only two instruction whose corresponding piece of code is not of constant complexity ($O(1)$), as they contain loops.

For the jump instructions, we do exactly what we would expect, we load the jump distance, add it to the register IN_1 , which we use to represent the VM program counter VPC. Incidentally, we can use the code for `jp` for the conditional jump `cjp`.

Control Instructions			
label	instruction	effect	comment
<code><jp></code>	<pre>MOVE IN_1 ACC STORE 0 LOADIN 1 1 ADD 0 MOVE ACC IN_1 JUMP $\langle jt \rangle$</pre>	<pre>$ACC := VPC$ $D(0) := ACC$ $ACC := D(VPC + 1)$ $ACC := ACC + D(0)$ $IN_1 := ACC$</pre>	<pre>cache VPC load i compute new VPC value update VPC jump back</pre>
<code><cjp></code>	<pre>dec IN_2 LOADIN 2 1 JUMP= <code><jp></code> MOVE IN_1 ACC ADDI 2 MOVE ACC IN_1 JUMP $\langle jt \rangle$</pre>	<pre>$SP := SP - 1$ $ACC := D(SP + 1)$ $VPC := VPC + 2$</pre>	<pre>update for pop pop value to ACC perform jump if $ACC = 0$ otherwise, go on point to next jump back</pre>

©:Michael Kohlhase 330

The imperative stack operations use the index register heavily. Note the use of the offset 8 in the `LOADIN`, this comes from the layout of VM that uses the bottom eight cells in the data store as a scratchpad.



Imperative Stack Operations: peek			
label	instruction	effect	comment
<code><peek></code>	<pre>MOVE IN_1 ACC STORE 0 LOADIN 1 1 MOVE ACC IN_1 inc IN_2 LOADIN 1 8 STOREIN 2 0 LOAD 0 ADDI 2 MOVE ACC IN_1 JUMP $\langle jt \rangle$</pre>	<pre>$ACC := IN_1$ $D(0) := ACC$ $ACC := D(VPC + 1)$ $IN_1 := ACC$ $ACC := D(IN_1 + 8)$ $ACC := D(0)$</pre>	<pre>cache VPC load i prepare push load $S(i)$ push $S(i)$ load old VPC compute new value update VPC jump back</pre>

©:Michael Kohlhase 331

The implementation for `poke` is quite similar, only that we use `STOREIN` instead of `LOADIN` at the strategic moment.

Imperative Stack Operations: poke			
-----------------------------------	--	--	--

label	instruction	effect	comment
⟨poke⟩	MOVE IN_1 ACC	$ACC := IN_1$	
	STORE 0	$D(0) := ACC$	cache VPC
	LOADIN 1 1	$ACC := D(VPC + 1)$	load i
	MOVE ACC IN_1	$IN_1 := ACC$	
	LOADIN 2 0	$ACC := S(i)$	pop to ACC
	STOREIN 1 8	$D(IN_1 + 8) := ACC$	store in $S(i)$
	dec IN_2	$IN_2 := IN_2 - 1$	
	LOAD 0	$ACC := D(0)$	get old VPC
	ADD 2	$ACC := ACC + 2$	add 2
	MOVE ACC IN_1		update VPC
	JUMP ⟨ jt ⟩		jump back


©: Michael Kohlhase
332


We have implemented the jump table for dispatch and all the $\mathcal{L}(\text{VM})$ instructions, which completes the VM virtual machine. As a consequence, we can run $\mathcal{L}(\text{VM})$ programs on REMA by loading them into program store and starting computation at the jump table start.



14.3 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 this¹

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 14.3.1** The **simple while language** SW is a simple programming languages with
 - ▷ **named variable** s (declare with `var <<name>> := <<exp>>`, assign with `<<name>> := <<exp>>`)
 - ▷ arithmetic/logic **expression** s with variables referenced by name
 - ▷ block-structured control structures (called **statement** s), e.g.
 - `while <<exp>> do <<statement>> end` and
 - `if <<exp>> then <<statement>> else <<statement>> end.`
 - ▷ **output** via `return <<exp>>`


©: Michael Kohlhase
333


To make the concepts involved concrete, we look at a concrete example.

¹Note 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.

Example: An SW Program for 12 Factorial

▷ Example 14.3.2 (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

```



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 “spaghetti code” induced by jump instructions (also called `goto`). See Edsger Dijkstra’s famous letter “Goto Considered Harmful” [Dij68] for a discussion.

Recall that we want to build 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 14.3.3 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 14.3.4 (Abstract SW Syntax)** We apply the abstract syntax to the SW program from Example 14.3.2:

<pre>var n:= 12; var a:= 1; while 2<=n do a:= a*n; n:= n-1 end return a</pre>	<pre>([("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")</pre>
--	--



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 $\mathcal{L}(\text{VM})$ program in concrete syntax. Instead we directly generate abstract SML expression.

We need an SML data type for $\mathcal{L}(\text{VM})$ programs, and an auxiliary function `wlen` that counts the numbers of instructions in a $\mathcal{L}(\text{VM})$ program (a list of instructions $\mathcal{L}(\text{VM})$ instructions). Fortunately, this is very simple.

An SML Data Type for $\mathcal{L}(\text{VM})$ 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
| jp      of noi     (* unconditional jump *)
| cjp     of noi     (* conditional jump *)
| peek   of index
| poke   of index
| halt

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
```

©: Michael Kohlhase 337

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 339, 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 14.3.5** An **environment** is a finite partial function from **key s** (identifiers) to values.
- ▷ We will need the following operations on environments:
 - ▷ creation of an empty environment (\leadsto the empty function)
 - ▷ insertion of a key/value pair $\langle k, v \rangle$ into an environment φ : ($\leadsto \varphi, [v/k]$)
 - ▷ lookup of the value v for a key k in φ ($\leadsto \varphi(k)$)
- ▷ Realization in SML by a structure with the following signature

```
type 'a env (* a is the value type *)
exception Unbound of id (* Unbound *)
val empty : 'a env
val insert : id * 'a * 'a env -> 'a env (* id is the key type *)
val lookup : id * 'a env -> 'a
```

©: Michael Kohlhase 338

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 $\mathcal{L}(\text{VM})$ instructions. From there, we only need to apply a simple conversion (which we omit) to numbers to obtain $\mathcal{L}(\text{VM})$ byte code.

Compiling SW programs

- ▷ SML function from SW programs (type `program`) to $\mathcal{L}(\text{VM})$ 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 (cde, env) = compileD(ds, empty, ~1)
  in
    cde @ compileS(s,env) @ compileE(e,env) @ [halt]
  end

```



©:Michael Kohlhase

339



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]

```



©:Michael Kohlhase

340



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 $\mathcal{L}(\text{VM})$ 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

```

```

end
| Seq ss => foldr (fn (s,c) => compileS(s,env) @ c) nil ss

```

 ©:Michael Kohlhase 341 



As we anticipated above, the `compileD` function is more complex than the other two. It gives $\mathcal{L}(\text{VM})$ 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

```

 ©:Michael Kohlhase 342 

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 $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$ with new operations next.

Note that the compiler we have seen above produces $\mathcal{L}(\text{VM})$ 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 14.3.6** Consider the following program that computes $12!$ and the corresponding $\mathcal{L}(\text{VM})$ program:

<pre> var n := 12; var a := 1; while 2 <= n do (a := a * n; n := n - 1;) return a; </pre>	<pre> 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 </pre>
--	--

 ©:Michael Kohlhase 

- ▷ 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 poking the stack position (must remember that)



©: Michael Kohlhase

343



14.4 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.

14.4.1 A Bare-Bones Language

We will now define a minimal core of the functional programming language *SML*, which we will call μ ML. 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, μ ML 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.

μ ML, 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 μ ML (micro ML)
 - ▷ no types: all values have type `int`, use `0` for `false` all other numbers for `true`.
- ▷ **Definition 14.4.1** **microML** μ ML is a simple functional programming languages with
 - ▷ **functional variable** `s` (declare and bind with `val <<name>> = <<exp>>`)
 - ▷ **named function** `s` (declare with `fun <<name>> (<<args>>) = <<exp>>`)
 - ▷ arithmetic/logic/control **expression** `s` with variables/functions referenced by name (no statements)



©: Michael Kohlhase

344



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 μ ML Program for 10 Squared

▷ Example 14.4.2 (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.

14.4.2 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 $\mathcal{L}(\text{VM})$

▷ **Definition 14.4.3** We obtain the language $\mathcal{L}(\text{VMP})$ by adding the following four commands to $\mathcal{L}(\text{VM})$:

- ▷ `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 μ ML function from Example 14.4.2 to $\mathcal{L}(\text{VMP})$. We have indicated corresponding parts of the programs by putting them on the same line.

A μ ML Program and its $\mathcal{L}(\text{VMP})$ Translation

▷ Example 14.4.4 (A μ ML Program and its $\mathcal{L}(\text{VMP})$ Translation)


```

[
proc 2 26,
con 0, arg 2, leq, cjp 5,
con 1, return,
con 1, arg 2, sub, arg 1,
call 0, arg 1, mul, return,
proc 0 6, con 10, return,

con 2, call 26, call 0,
halt]

```

```

let
fun exp(x,n) =
  if n<=0
  then 1
  else x*exp(x,n-1)

val y 10
in
  exp(y,2)
end

```



To see how these four commands together can simulate procedures, we simulate the program from the last slide, keeping track of the stack.

Static Procedures (Simulation)

Example 14.4.5 We go through the $\mathcal{L}(\text{VMP})$ 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]` 2

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]
```

30	0
2	

- ▷ call pushes the return address (of the call statement in the $\mathcal{L}(\text{VM})$ 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]
```

10	
30	0
2	

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 $\mathcal{L}(\text{VM})$ 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
10
2

0
-1
-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
10
2

0
-1
-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
10
2

0
-1
-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
10
2

0
-1
-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
10
2

0
-1
-2

we subtract

▷	<pre>[proc 2 26, con 0, arg 2, leq, cjp 5, con 1, return, con 1, arg 2, sub, <u>arg 1</u>, call 0, arg 1, mul, return, proc 0 6, con 10, return, con 2, call 26, call 0, halt]</pre>	<table style="border-collapse: collapse; margin: auto;"> <tr><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">1</td><td></td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">38</td><td style="padding: 0 5px;">0</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td style="padding: 0 5px;">-1</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">2</td><td style="padding: 0 5px;">-2</td></tr> </table>	10		1		38	0	10	-1	2	-2
10												
1												
38	0											
10	-1											
2	-2											

then we push the second argument (from the call frame position -1)

▷	<pre>[proc 2 26, con 0, arg 2, leq, cjp 5, con 1, return, con 1, arg 2, sub, arg 1, <u>call 0</u>, arg 1, mul, return, proc 0 6, con 10, return, con 2, call 26, call 0, halt]</pre>	<table style="border-collapse: collapse; margin: auto;"> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">22</td><td style="padding: 0 5px;">0</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td style="padding: 0 5px;">-1</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">1</td><td style="padding: 0 5px;">-2</td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">38</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">2</td><td></td></tr> </table>	22	0	10	-1	1	-2	38		10		2	
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.

<pre>[proc 2 26, <u>con 0, arg 2</u>, 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]</pre>	<table style="border-collapse: collapse; margin: auto;"> <tr><td style="border: 1px solid gray; padding: 2px 10px;">1</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">0</td><td></td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">22</td><td style="padding: 0 5px;">0</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td style="padding: 0 5px;">-1</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">1</td><td style="padding: 0 5px;">-2</td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">38</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">2</td><td></td></tr> </table>	1		0		22	0	10	-1	1	-2	38		10		2	
1																	
0																	
22	0																
10	-1																
1	-2																
38																	
10																	
2																	

we augment the stack

▷	<pre>[proc 2 26, con 0, arg 2, <u>leq, cjp 5</u>, 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]</pre>	<table style="border-collapse: collapse; margin: auto;"> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">22</td><td style="padding: 0 5px;">0</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td style="padding: 0 5px;">-1</td></tr> <tr style="background-color: #e0e0e0;"><td style="border: 1px solid gray; padding: 2px 10px;">1</td><td style="padding: 0 5px;">-2</td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">38</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">10</td><td></td></tr> <tr><td style="border: 1px solid gray; padding: 2px 10px;">2</td><td></td></tr> </table>	22	0	10	-1	1	-2	38		10		2	
22	0													
10	-1													
1	-2													
38														
10														
2														

compare the top two, and jump ahead (on false)

▷	<pre>[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]</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td></td></tr> <tr><td>1</td><td></td></tr> <tr style="background-color: #cccccc;"><td>22</td><td>0</td></tr> <tr style="background-color: #cccccc;"><td>10</td><td>-1</td></tr> <tr style="background-color: #cccccc;"><td>1</td><td>-2</td></tr> <tr><td>38</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>2</td><td></td></tr> </table>	1		1		22	0	10	-1	1	-2	38		10		2	
1																		
1																		
22	0																	
10	-1																	
1	-2																	
38																		
10																		
2																		

we augment the stack again

▷	<pre>[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]</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>10</td><td></td></tr> <tr><td>0</td><td></td></tr> <tr style="background-color: #cccccc;"><td>22</td><td>0</td></tr> <tr style="background-color: #cccccc;"><td>10</td><td>-1</td></tr> <tr style="background-color: #cccccc;"><td>1</td><td>-2</td></tr> <tr><td>38</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>2</td><td></td></tr> </table>	10		0		22	0	10	-1	1	-2	38		10		2	
10																		
0																		
22	0																	
10	-1																	
1	-2																	
38																		
10																		
2																		

subtract and push the first argument

▷	<pre>[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]</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr style="background-color: #cccccc;"><td>22</td><td>0</td></tr> <tr style="background-color: #cccccc;"><td>10</td><td>-1</td></tr> <tr style="background-color: #cccccc;"><td>0</td><td>-2</td></tr> <tr><td>22</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>1</td><td></td></tr> <tr><td>38</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>2</td><td></td></tr> </table>	22	0	10	-1	0	-2	22		10		1		38		10		2	
22	0																			
10	-1																			
0	-2																			
22																				
10																				
1																				
38																				
10																				
2																				

call pushes the return address and moves the current frame up

▷	<pre>[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]</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td></td></tr> <tr><td>0</td><td></td></tr> <tr style="background-color: #cccccc;"><td>22</td><td>0</td></tr> <tr style="background-color: #cccccc;"><td>10</td><td>-1</td></tr> <tr style="background-color: #cccccc;"><td>0</td><td>-2</td></tr> <tr><td>22</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>1</td><td></td></tr> <tr><td>38</td><td></td></tr> <tr><td>10</td><td></td></tr> <tr><td>2</td><td></td></tr> </table>	0		0		22	0	10	-1	0	-2	22		10		1		38		10		2	
0																								
0																								
22	0																							
10	-1																							
0	-2																							
22																								
10																								
1																								
38																								
10																								
2																								

we augment the stack again,

		22	0
		10	-1
		0	-2
▷	[proc 2 26,		
	con 0, arg 2, leq, cjp 5,		
	con 1, return,	22	
	con 1, arg 2, sub, arg 1,	10	
	call 0, arg 1, mul, return,	1	
	proc 0 6, con 10, return,	38	
	con 2, call 26, call 0,	10	
	halt]	2	

leq compares the top two numbers, cjp pops the result and does not jump.

		1	
		22	0
		10	-1
		0	-2
▷	[proc 2 26,		
	con 0, arg 2, leq, cjp 5,		
	con 1, return,	22	
	con 1, arg 2, sub, arg 1,	10	
	call 0, arg 1, mul, return,	1	
	proc 0 6, con 10, return,	38	
	con 2, call 26, call 0,	10	
	halt]	2	

we push the result value 1

		1	
		22	0
		10	-1
		1	-2
▷	[proc 2 26,		
	con 0, arg 2, leq, cjp 5,		
	con 1, return,	22	
	con 1, arg 2, sub, arg 1,	10	
	call 0, arg 1, mul, return,	1	
	proc 0 6, con 10, return,	38	
	con 2, call 26, call 0,	10	
	halt]	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.

		10	
		1	
		22	0
		10	-1
		1	-2
	[proc 2 26,		
	con 0, arg 2, leq, cjp 5,		
	con 1, return,	22	
	con 1, arg 2, sub, arg 1,	10	
	call 0, arg 1, mul, return,	1	
	proc 0 6, con 10, return,	38	
	con 2, call 26, call 0,	10	
	halt]	2	

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, <u>mul</u> , return, proc 0 6, con 10, return, con 2, call 26, call 0, halt]	10	
		22	0
		10	-1
		1	-2
		38	
		10	
		2	

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, <u>mul</u> , <u>return</u> , proc 0 6, con 10, return, con 2, call 26, call 0, halt]	10	
		38	0
		10	-1
		2	-2

- ▷ 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, <u>arg 1</u> , <u>mul</u> , return, con 2, call 26, call 0, halt]	100	
		38	0
		10	-1
		2	-2

we push argument 1 (in this case 10), multiply the top two numbers, and push the result to the stack

▷	[proc 2 26, con 0, arg 2, leq, cjp 5, con 1, return, con 1, arg 2, sub, arg 1, call 0, arg 1, <u>mul</u> , <u>return</u> , proc 0 6, con 10, return, con 2, call 26, call 0, halt]	100
---	---	-----

- ▷ 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]
```

100

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 $\mathcal{L}(\text{VMP})$ 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 on the stack)

`call p` pushes the current program address (opens a new frame), and jumps to the program address p

`return` takes the current frame from the stack, jumps to previous program address. (which is cached in the frame)

- ▷ `call` and `return` jointly have the effect of replacing the arguments by the result of the procedure.

- ▷ 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 14.4.6** We obtain the **virtual machine with procedures** VMP by extending VM by ASM implementations for the new $\mathcal{L}(\text{VMP})$ instructions. (on the next slides)

©: Michael Kohlhase
350

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 IN_1 ACC	$ACC := VPC$	
	STORE 0	$D(0) := ACC$	cache VPC
	LOADIN 1 2	$ACC := D(VPC + 2)$	load length
	ADD 0	$ACC := ACC + D(0)$	compute new VPC value
	MOVE ACC IN_1	$IN_1 := ACC$	update VPC
	JUMP ⟨jt⟩		jump back

©: Michael Kohlhase
351

Realizing arg

- ▷ `arg i` pushes the i^{th} argument from the current frame to the stack.
- ▷ use the register IN_3 for the frame pointer. (extend for first frame)

label	instruction	effect	comment
⟨arg⟩	LOADIN 1 1	$ACC := D(VPC + 1)$	load i
	STORE 0	$D(0) := ACC$	cache i
	MOVE IN_3 ACC		
	STORE 1	$D(1) := FP$	cache FP
	ADDI - 1		
	SUB 0	$ACC := FP - 1 - i$	load argument position
	MOVE ACC IN_3	$FP := ACC$	move it to FP
	inc IN_2	$SP := SP + 1$	prepare push
	LOADIN 3 0	$ACC := D(FP)$	load arg i
	STOREIN 2 0	$D(SP) := ACC$	push arg i
	LOAD 1	$ACC := D(1)$	load FP
	MOVE ACC IN_3	$FP := ACC$	recover FP
	MOVE IN_1 ACC		
	ADDI 2		
	MOVE ACC IN_1	$VPC := VPC + 2$	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 IN_1 ACC		cache current VPC
	STORE 0	$D(0) := IN_1$	prepare push for later
	inc IN_2	$SP := SP + 1$	load argument
	LOADIN 1 1	$ACC := D(VPC + 1)$	add displacement and skip proc $a l$
	ADDI $2^{24} + 3$	$ACC := ACC + 2^{24} + 3$	point to the first instruction
	MOVE ACC IN_1	$VPC := ACC$	stealing a from proc $a l$
	LOADIN 1 - 2	$ACC := D(VPC - 2)$	push the number of arguments
	STOREIN 2 0	$D(SP) := ACC$	prepare push
	inc IN_2	$SP := SP + 1$	load FP
	MOVE IN_3 ACC	$ACC := IN_3$	create anchor cell
	STOREIN 2 0	$D(SP) := ACC$	update FP
	MOVE IN_2 IN_3	$FP := SP$	prepare push
	inc IN_2	$SP := SP + 1$	load VPC
	LOAD 0	$ACC := D(0)$	point to next instruction
	ADDI 2	$ACC := ACC + 2$	push the return address
	STOREIN 2 0	$D(SP) := ACC$	jump back
	JUMP ⟨ jt ⟩		



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	$ACC := D(SP)$	load top value
	STORE 0	$D(0) := ACC$	cache it
	LOADIN 2 - 1	$ACC := D(SP - 1)$	load return address
	MOVE ACC IN_1	$IN_1 := ACC$	set VPC to it
	LOADIN 3 - 1	$ACC := D(FP - 1)$	load the number n of arguments
	STORE 1	$D(1) := D(FP - 1)$	cache it
	MOVE IN_3 ACC	$ACC := FP$	$ACC = FP$
	ADDI - 1	$ACC := ACC - 1$	$ACC = FP - 1$
	SUB 1	$ACC := ACC - D(1)$	$ACC = FP - 1 - n$
	MOVE ACC IN_2	$IN_2 := ACC$	$SP = ACC$
	LOADIN 3 0	$ACC := D(FP)$	load anchor value
	MOVE ACC IN_3	$IN_3 := ACC$	point to previous frame
	LOAD 0	$ACC := D(0)$	load cached return value
	STOREIN 2 0	$D(IN_2) := ACC$	pop return value
	JUMP (jt)		jump back


©:Michael Kohlhase
354


Note that all the realizations of the $\mathcal{L}(\text{VMP})$ 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 $\mathcal{L}(\text{VMP})$ programs. Just as above, we will write this compiler in SML.

14.4.3 Compiling Basic Functional Programs

For the μML compiler we will proceed as above: we first introduce SML data types for the abstract syntax of $\mathcal{L}(\text{VMP})$ and μML and then we define a SML function that converts abstract μML programs to abstract $\mathcal{L}(\text{VMP})$ 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

```


©:Michael Kohlhase
355


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 intuition by looking at a concrete example: the μML function from Example 14.4.2.

Concrete vs. Abstract Syntax of μ ML

- ▷ A μ ML 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 $\mathcal{L}(\text{VMP})$ as an extension of the one for $\mathcal{L}(\text{VM})$. 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 $\mathcal{L}(\text{VMP})$

- ▷ Extensions to the abstract data types for $\mathcal{L}(\text{VM})$...

```

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 μ ML 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 μ ML: 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)
```



©: Michael Kohlhase

358



As μ ML 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 μ ML

```
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)
```



©: Michael Kohlhase

359



Next we define the μ ML expression compiler: a function `compileE` that compiles abstract μ ML expressions into lists of abstract $\mathcal{L}(\text{VMP})$ 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 μ ML 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 $\mathcal{L}(\text{VMP})$ 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 μ ML: 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
```

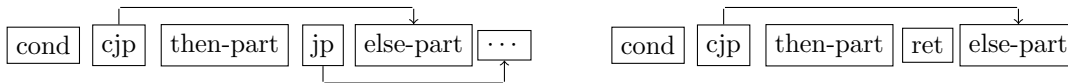
```

      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 `compileE`. 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 need 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.

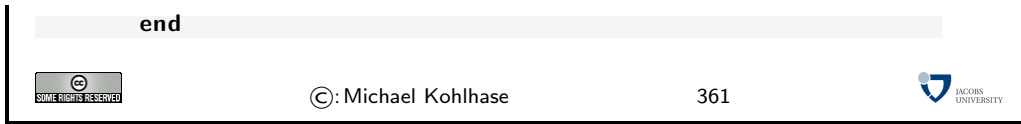
The declaration compiler recurses over the list of declarations, and for each declaration, it first inserts the procedure name into the environment `env`, yielding a new environment `env'`, into which we then insert the argument list via an auxiliary function `insertArgs` we have defined before. In the 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 μ ML 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 $\mathcal{L}(\text{VMP})$ 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 $\mathcal{L}(\text{VMP})$ compiled from μML .

Note that even if we do not use them in the μML compiler, `peek` and `poke` are still available in $\mathcal{L}(\text{VMP})$ (which is defined to be a superset of $\mathcal{L}(\text{VM})$). 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 $\mathcal{L}(\text{VMP})$ code from μML programs.
- ▷ μML is minimal, but Turing-Complete (has conditionals and procedures)

©: Michael Kohlhase
362

14.5 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, . . .

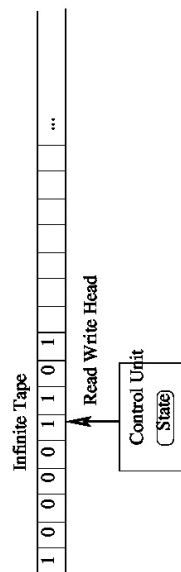
- ▷ **Conjecture 14.5.1** [Church/Turing] *(unprovable, but accepted)*
All non-trivial machine models and programming languages are equivalent



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.”

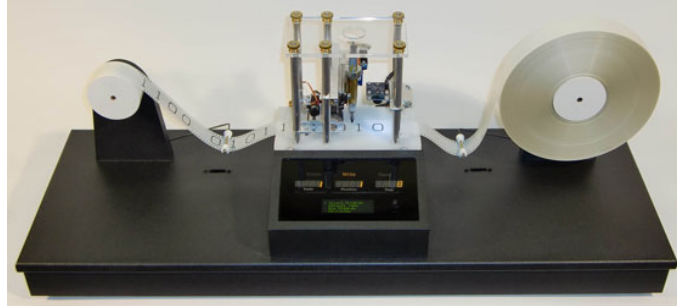


Note that the physical realization of the machine as a box with a (paper) tape is immaterial, it is inspired by the technology at the time of its inception (in the late 1940ies; the age of ticker-tape communication).

A Physical Realization of a Turing Machine

▷ **Note:** Turing machine can be built, but that is not the important aspect

▷ **Example 14.5.2 (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)



©: Michael Kohlhase

365



To use (i.e. simulate) Turing machines, we have to make the notion a bit more precise.

Turing Machine: The Definition

▷ **Definition 14.5.3** 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 special start state)
- ▷ An **action table** that tells the machine what symbol to write, how to move the head and what its new state will be, given the symbol it has just read on the tape and the state it is currently in. (If no entry applicable the machine will halt)

▷ and now again, mathematically:

▷ **Definition 14.5.4** 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 **state** s , $s_0 \in \mathcal{S}$ is the **initial state**, $\mathcal{F} \subseteq \mathcal{S}$ is the set of **final state** s , and \mathcal{R} is a function $\mathcal{R}: \mathcal{S} \setminus \mathcal{F} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathcal{A} \times \{R, L\}$ 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.



©: Michael Kohlhase

366



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 14.5.3 are the alphabet used for data representation on the tape, the set of states, the initial state, and the **action**table; so they are what we have to give to specify a Turing machine.

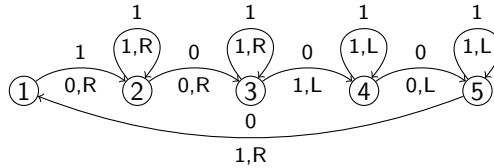
Turing Machine

Example 14.5.5 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 1 1
6	s_5	0 1 0 1	14	s_5	1 0 0 1 1
7	s_5	0 1 0 1	15	s_1	11 0 1 1
8	s_1	1 1 0 1		— 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 s_1 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 14.5.6** ([Church/Turing]) (unprovable, but accepted)
Anything that can be computed at all, can be computed by a Turing Machine
- ▷ **Definition 14.5.7** 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 Section 14.0.

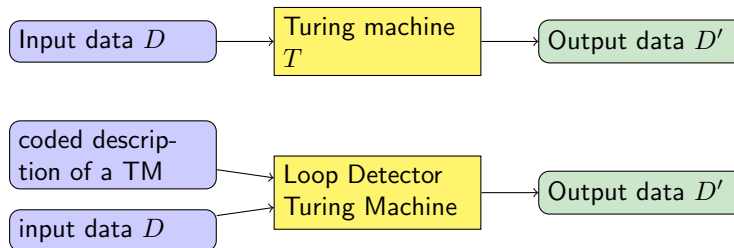
Universal Turing machines

- ▷ **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 14.5.8** *Such a Turing machine is indeed possible (e.g. with 2 states, 18 symbols)*

▷ **Definition 14.5.9** 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 $\mathcal{L}(\text{VMP})$ and SW to $\mathcal{L}(\text{VM})$ 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 $\mathcal{L}(\text{VM})$, then we could not compile the control structures of SW or μML into $\mathcal{L}(\text{VM})$ or $\mathcal{L}(\text{VMP})$. So we should read the Church/Turing hypothesis as a statement of equivalence of all non-trivial programming languages.

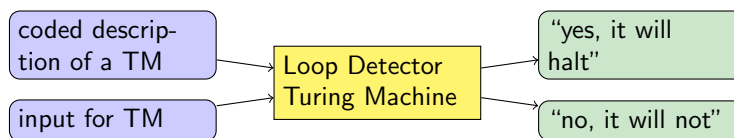
Question: So, if all non-trivial programming languages can compute the same, are there things that none of them can compute? This is what we will have a look at next.

▷ Is there anything that cannot be computed by a TM? [Tur36]

▷ **Theorem 14.5.10 (Halting Problem)** *No Turing machine can infallibly tell if another Turing machine will get stuck in an infinite loop on some given input.*

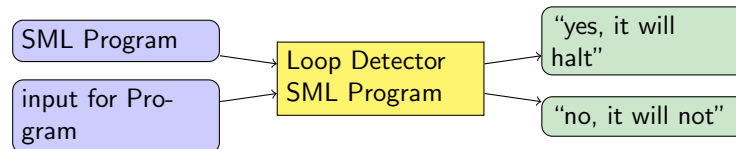
▷ The problem of determining whether a Turing machine will halt on an input is called the **halting problem**.

▷



▷ **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
<pre>fun halter (s) = ""; halter : string -> string fun looper (s) = looper(s); looper : string -> string</pre>	<pre>will_halt(halter,""); val true : bool will_halt(looper,""); val false : bool</pre>

P.2 Consider the following program

```
fun turing (prog) =
  if will_halt(eval(prog),prog) then looper("") else "";
turing : string -> string
```

P.3 Yeah, so what? what happens, if we feed the turing function to itself?

 ©:Michael Kohlhase 372 

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("") else "";
```



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

 ©:Michael Kohlhase 373 

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.

Chapter 15

The Information and Software Architecture of the Internet and World Wide Web

In the last chapters we have seen how to build computing devices, and how to program them with high-level programming languages. But this is only part of the computing infrastructure we have gotten used to in the last two decades: computers are nowadays globally networked on the Internet, and we use computation on remote computers and information services on the World Wide Web on a day-to-day basis.

In this chapter we will look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up.

15.1 Overview

We start off with a disambiguation of the concepts of Internet and World Wide Web that are often used interchangeably (and thus imprecisely) in the popular discussion. In fact, the form quite different pieces in the general networking infrastructure, with the World Wide Web building on the Internet as one of many services. We will give an overview over the devices and protocols driving the Internet in Section 15.1 and on the central concepts of the World Wide Web in Section 15.2.

The Internet and the Web

- ▷ **Definition 15.1.1** The **Internet** is a worldwide computer network that connects hundreds of thousands of smaller networks. (The mother of all networks)
- ▷ **Definition 15.1.2** The **World Wide Web (WWW)** is the interconnected system of servers that support multimedia documents, i.e. the multimedia part of the Internet.
- ▷ The Internet and WWW form critical infrastructure for modern society and commerce.
- ▷ The Internet/WWW is huge:

Year	Web	Deep Web	eMail
1999	21 TB	100 TB	11TB
2003	167 TB	92 PB	447 PB
2010	????	?????	?????

▷ We want to understand how it works (services and scalability issues)





One of the central things to understand about the Internet and the WWW is that they have been growing exponentially over the last decades in terms of traffic and available content. In fact, we do not really know how big the Internet/WWW are, its distributed, and increasingly commercial nature and global scale make that increasingly difficult to measure.

Of course, we also want to understand the units used in the measurement of the size of the Internet, this is next.

How much Information?

Bit (b)	<i>binary digit 0/1</i>
Byte (B)	<i>8 bit</i>
2 Bytes	A Unicode character in UTF.
10 Bytes	your name.
Kilobyte (kB)	<i>1,000 bytes OR 10^3 bytes</i>
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (MB)	<i>1,000,000 bytes OR 10^6 bytes</i>
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)	<i>1,000,000,000 bytes or 10^9 bytes</i>
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)	<i>1,000,000,000,000 bytes or 10^{12} bytes</i>
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)	<i>1,000,000,000,000,000 bytes or 10^{15} bytes</i>
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)	<i>1,000,000,000,000,000,000 bytes or 10^{18} bytes</i>
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)	<i>1,000,000,000,000,000,000,000 bytes or 10^{21} bytes</i>
2 Zettabytes	Total volume digital data transmitted in 2011
100 Zettabytes	Data equivalent to the human Genome in one body.


©: Michael Kohlhase
375


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.

A Timeline of the Internet and the Web

- ▷ Early 1960s: introduction of the network concept
- ▷ 1970: ARPANET, scholarly-aimed networks
- ▷ 62 computers in 1974
- ▷ 1975: Ethernet developed by Robert Metcalfe
- ▷ 1980: TCP/IP
- ▷ 1982: The first computer virus, Elk Cloner, spread via Apple II floppy disks
- ▷ 500 computers in 1983
- ▷ 28,000 computers in 1987
- ▷ 1989: Web invented by Tim Berners-Lee
- ▷ 1990: First Web browser based on HTML developed by Berners-Lee
- ▷ Early 1990s: Andreessen developed the first graphical browser (Mosaic)
- ▷ 1993: The US White House launches its Web site

▷ 1993 -: commercial/public web explodes



©: Michael Kohlhase

376



We will now look at the information and software architecture of the Internet and the World Wide Web (WWW) from the ground up.

15.2 Internet Basics

We will show aspects of how the Internet can cope with this enormous growth of numbers of computers, connections and services.

The growth of the Internet rests on three design decisions taken very early on. The Internet

- 1) is a packet-switched network rather than a network, where computers communicate via dedicated physical communication lines.
- 2) is a network, where control and administration are decentralized as much as possible.
- 3) is an infrastructure that only concentrates on transporting packets/datagrams between computers. It does not provide special treatment to any packets, or try to control the content of the packets.

The first design decision is a purely technical one that allows the existing communication lines to be shared by multiple users, and thus save on hardware resources. The second decision allows the administrative aspects of the Internet to scale up. Both of these are crucial for the scalability of the Internet. The third decision (often called “net neutrality”) is hotly debated. The defenders cite that net neutrality keeps the Internet an open market that fosters innovation, where as the attackers say that some uses of the network (illegal file sharing) disproportionately consume resources.

Package-Switched Networks

▷ **Definition 15.2.1** A **packet-switched network** divides messages into small **network packets** that are transported separately and re-assembled at the target.

▷ **Advantages:**

- ▷ many users can share the same physical communication lines.
- ▷ packets can be routed via different paths. (bandwidth utilization)
- ▷ bad packets can be re-sent, while good ones are sent on. (network reliability)
- ▷ packets can contain information about their sender, destination.
- ▷ no central management instance necessary (scalability, resilience)



©: Michael Kohlhase

377



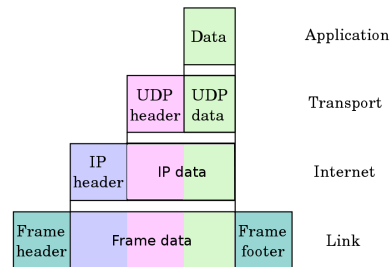
These ideas are implemented in the Internet Protocol Suite, which we will present in the rest of the section. A main idea of this set of protocols is its layered design that allows to separate concerns and implement functionality separately.

The Internet Protocol Suite

- ▷ **Definition 15.2.2** The **Internet Protocol Suite** (commonly known as **TCP/IP**) is the set of communications protocols used for the Internet and other similar networks. It structured into 4 layers.

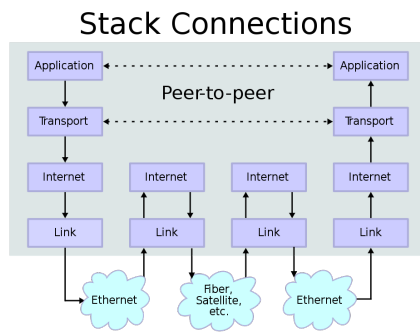
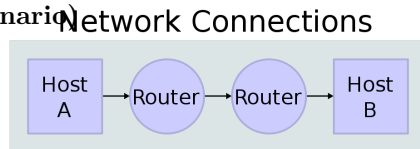
Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	UDP, TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

- ▷ **Layers in TCP/IP:** TCP/IP uses encapsulation to provide abstraction of protocols and services.
- ▷ An application (the highest level of the model) uses a set of protocols to send its data down the layers, being further encapsulated at each level.



The Internet as a Network of Networks

- ▷ **Example 15.2.3 (TCP/IP Scenario)** Consider a situation with two Internet host computers communicate across local network boundaries.
- ▷ network boundaries are constituted by internetworking gateways (routers).
- ▷ **Definition 15.2.4** A **router** is a purposely customized computer used to forward data among computer networks beyond directly connected devices.
- ▷ A router implements the link and internet layers only and has two network connections.



We will now take a closer look at each of the layers shown above, starting with the lowest one. Instead of going into network topologies, protocols, and their implementation into physical signals that make up the link layer, we only discuss the devices that deal with them. Network Interface controllers are specialized hardware that encapsulate all aspects of link-level communication, and we take them as black boxes for the purposes of this course.

Network Interfaces

- ▷ The nodes in the Internet are computers, the edges communication channels
- ▷ **Definition 15.2.5** A **network interface controller (NIC)** is a hardware device that handles an interface to a computer network and thus allows a network-

capable device to access that network.

- ▷ **Definition 15.2.6** Each NIC contains a unique number, the **media access control address (MAC address)**, identifies the device uniquely on the network.
- ▷ MAC addresses are usually 48-bit numbers issued by the manufacturer, they are usually displayed to humans as six groups of two hexadecimal digits, separated by hyphens (-) or colons (:), in transmission order, e.g. 01-23-45-67-89-AB, 01:23:45:67:89:AB.

Definition 15.2.7 A **network interface** is a software component in the operating system that implements the higher levels of the network protocol (the NIC handles the lower ones).

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

- ▷ A computer can have more than one network interface. (e.g. a router)



The next layer is the Internet Layer, it performs two parts: addressing and packing packets.

Internet Protocol and IP Addresses

- ▷ **Definition 15.2.8** The **Internet Protocol (IP)** is a protocol used for communicating data across a packet-switched internetwork. The Internet Protocol defines addressing methods and structures for datagram encapsulation. The Internet Protocol also routes data packets between networks
- ▷ **Definition 15.2.9** An Internet Protocol (IP) address is a numerical label that is assigned to devices participating in a computer network, that uses the Internet Protocol for communication between its nodes.
- ▷ An IP address serves two principal functions: host or network interface identification and location addressing.
- ▷ **Definition 15.2.10** The global IP address space allocations are managed by the **Internet Assigned Numbers Authority (IANA)**, delegating allocate IP address blocks to five Regional Internet Registries (RIRs) and further to Internet service providers (ISPs).
- ▷ **Definition 15.2.11** The Internet mainly uses **Internet Protocol Version 4 (IPv4)** [RFC80], which uses 32-bit numbers (**IPv4 address es**) for identification of network interfaces of Computers.
- ▷ IPv4 was standardized in 1980, it provides 4,294,967,296 (2^{32}) possible unique addresses. With the enormous growth of the Internet, we are fast running out of IPv4 addresses
- ▷ **Definition 15.2.12** **Internet Protocol Version 6 (IPv6)** [DH98], which uses 128-bit numbers (**IPv6 address es**) for identification.

- ▷ Although IP addresses are stored as binary numbers, they are usually displayed in human-readable notations, such as 208.77.188.166 (for IPv4), and 2001 : db8 : 0 : 1234 : 0 : 567 : 1 : 1 (for IPv6).



The Internet infrastructure is currently undergoing a dramatic retooling, because we are moving from IPv4 to IPv6 to counter the depletion of IP addresses. Note that this means that all routers and switches in the Internet have to be upgraded. At first glance, it would seem that that this problem could have been avoided if we had only anticipated the need for more the 4 million computers. But remember that TCP/IP was developed at a time, where the Internet did not exist yet, and it's precursor had about 100 computers. Also note that the IP addresses are part of every packet, and thus reserving more space for them would have wasted bandwidth in a time when it was scarce.

We will now go into the detailed structure of the IP packets as an example of how a low-level protocol is structured. Basically, an IP packet has two parts: the “header”, whose sequence of bytes is strictly standardized, and the “payload”, a segment of bytes about which we only know the length, which is specified in the header.

The Structure of IP Packets

- ▷ **Definition 15.2.13** IP packets are composed of a 160b header and a payload. The IPv4 packet header consists of:

b	name	comment
4	version	IPv4 or IPv6 packet
4	Header Length	in multiples 4 bytes (e.g., 5 means 20 bytes)
8	QoS	Quality of Service, i.e. priority
16	length	of the packet in bytes
16	fragid	to help reconstruct the packet from fragments,
3	fragmented	DF $\hat{=}$ “Don't fragment” / MF $\hat{=}$ “More Fragments”
13	fragment offset	to identify fragment position within packet
8	TTL	Time to live (router hops until discarded)
8	protocol	TCP, UDP, ICMP, etc.
16	Header Checksum	used in error detection,
32	Source IP	
32	target IP	
...	optional flags	according to header length

- ▷ Note that delivery of IP packets is not guaranteed by the IP protocol.



As the internet protocol only supports addressing, routing, and packaging of packets, we need another layer to get services like the transporting of files between specific computers. Note that the IP protocol does not guarantee that packets arrive in the right order or indeed arrive at all, so the transport layer protocols have to take the necessary measures, like packet re-sending or handshakes, ...

The Transport Layer

- ▷ **Definition 15.2.14** The **transport layer** is responsible for delivering data to the appropriate application process on the host computers by forming data packets, and adding source and destination port numbers in the header.
- ▷ **Definition 15.2.15** The internet protocol mainly uses suite the **Transmission Control Protocol (TCP)** and **User Datagram Protocol (UDP)** protocols at the transport layer.
- ▷ TCP is used for communication, UDP for multicasting and broadcasting.
- ▷ TCP supports virtual circuits, i.e. provide connection oriented communication over an underlying packet oriented datagram network. (hide/reorder packets)
- ▷ TCP provides end-to-end reliable communication (error detection & automatic repeat)



©: Michael Kohlhase

383

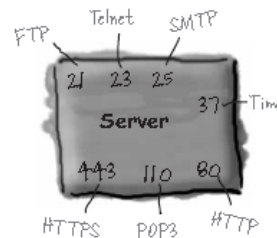


We will see that there are quite a lot of services at the network application level. And indeed, many web-connected computers run a significant subset of them at any given time, which could lead to problems of determining which packets should be handled by which service. The answer to this problem is a system of “ports” (think pigeon holes) that support finer-grained addressing to the various services.

Ports

- ▷ **Definition 15.2.16** To separate the services and protocols of the network application layer, network interfaces assign them specific **port**, referenced by a number.
- ▷ **Example 15.2.17** We have the following ports in common use on the Internet

Port	use	comment
22	SSH	remote shell
53	DNS	Domain Name System
80	HTTP	World Wide Web
443	HTTPS	HTTP over SSL



©: Michael Kohlhase

384



On top of the transport-layer services, we can define even more specific services. From the perspective of the internet protocol suite this layer is unregulated, and application-specific. From a user perspective, many useful services are just “applications” and live at the application layer.

The Application Layer

▷ **Definition 15.2.18** The **application layer** of the internet protocol suite contains all protocols and methods that fall into the realm of process-to-process communications via an Internet Protocol (IP) network using the Transport Layer protocols to establish underlying host-to-host connections.

▷ **Example 15.2.19 (Some Application Layer Protocols and Services)**

BitTorrent	Peer-to-peer	Atom	Syndication
DHCP	Dynamic Host Configuration	DNS	Domain Name System
FTP	File Transfer Protocol	HTTP	HyperText Transfer
IMAP	Internet Message Access	IRC	Internet Relay Chat
NFS	Network File System	NNTP	Network News Transfer
NTP	Network Time Protocol	POP	Post Office Protocol
RPC	Remote Procedure Call	SMB	Server Message Block
SMTP	Simple Mail Transfer	SSH	Secure Shell
TELNET	Terminal Emulation	WebDAV	Write-enabled Web



©: Michael Kohlhase

385



We will now go into some of the most salient services on the network application layer.

The domain name system is a sort of telephone book of the Internet that allows us to use symbolic names for hosts like `kwarc.info` instead of the IP number 212.201.49.189.

Domain Names

▷ **Definition 15.2.20** The **DNS (Domain Name System)** is a distributed set of servers that provides the mapping between (static) IP addresses and domain names.

▷ **Example 15.2.21** e.g. `www.kwarc.info` stands for the IP address 212.201.49.189.

▷ **Definition 15.2.22** Domain names are hierarchically organized, with the most significant part (the **top-level domain TLD**) last.

▷ networked computers can have more than one DNS name. (**virtual servers**)

▷ Domain names must be registered to ensure uniqueness (**registration fees vary, cybersquatting**)

▷ **Definition 15.2.23 ICANN** is a non-profit organization was established to regulate human-friendly domain names. It approves top-level domains, and corresponding domain name registrars and delegates the actual registration to them.



©: Michael Kohlhase

386



Let us have a look at a selection of the top-level domains in use today.

Domain Name Top-Level Domains

▷ `.com` ("commercial") is a generic top-level domain. It was one of the original top-level domains, and has grown to be the largest in use.

▷ `.org` ("organization") is a generic top-level domain, and is mostly associated with non-profit organizations. It is also used in the charitable field, and used by the open-source movement. Government sites and Political parties in the

US have domain names ending in .org

- ▷ .net (“network”) is a generic top-level domain and is one of the original top-level domains. Initially intended to be used only for network providers (such as Internet service providers). It is still popular with network operators, it is often treated as a second .com. It is currently the third most popular top-level domain.
- ▷ .edu (“education”) is the generic top-level domain for educational institutions, primarily those in the United States. One of the first top-level domains, .edu was originally intended for educational institutions anywhere in the world. Only post-secondary institutions that are accredited by an agency on the U.S. Department of Education’s list of nationally recognized accrediting agencies are eligible to apply for a .edu domain.
- ▷ .info (“information”) is a generic top-level domain intended for informative website’s, although its use is not restricted. It is an unrestricted domain, meaning that anyone can obtain a second-level domain under .info. The .info was one of many extension(s) that was meant to take the pressure off the overcrowded .com domain.
- ▷ .gov (“government”) a generic top-level domain used by government entities in the United States. Other countries typically use a second-level domain for this purpose, e.g., .gov.uk for the United Kingdom. Since the United States controls the .gov Top Level Domain, it would be impossible for another country to create a domain ending in .gov.
- ▷ .biz (“business”) the name is a phonetic spelling of the first syllable of “business”. A generic top-level domain to be used by businesses. It was created due to the demand for good domain names available in the .com top-level domain, and to provide an alternative to businesses whose preferred .com domain name which had already been registered by another.
- ▷ .xxx (“porn”) the name is a play on the verdict “X-rated” for movies. A generic top-level domain to be used for sexually explicit material. It was created in 2011 in the hope to move sexually explicit material from the “normal web”. But there is no mandate for porn to be restricted to the .xxx domain, this would be difficult due to problems of definition, different jurisdictions, and free speech issues.



Note: Anybody can register a domain name from a registrar against a small yearly fee. Domain names are given out on a first-come-first-serve basis by the domain name registrars, which usually also offer services like domain name parking, DNS management, URL forwarding, etc.

The next application-level service is the SMTP protocol used for sending e-mail. It is based on the telnet protocol for remote terminal emulation which we do not discuss here.

telnet is one of the oldest protocols, which uses TCP directly to send text-based messages between a terminal client (on the local host) and a terminal server (on the remote host). The operation of a remote terminal is the following: the terminal server on the remote host receives commands from the terminal client on the local host, executes them on the remote host and sends back the results to the client on the local host.



A Protocol Example: SMTP over telnet

▷ We call up the telnet service on the Jacobs mail server
telnet exchange.jacobs-university.de 25

▷ it identifies itself (have some patience, it is very busy)
Trying 10.70.0.128...
Connected to exchange.jacobs-university.de.
Escape character is '^]'.
220 SHUBCAS01.jacobs.jacobs-university.de
Microsoft ESMTMP MAIL Service ready at Tue, 3 May 2011 13:51:23 +0200

▷ We introduce ourselves politely (but we lie about our identity)
helo mailhost.domain.tld

▷ It is really very polite.
250 SHUBCAS04.jacobs.jacobs-university.de Hello [10.222.1.5]

 ©: Michael Kohlhase 388 

SMTP over telnet: The e-mail itself

▷ We start addressing an e-mail (again, we lie about our identity)
mail from: user@domain.tld

▷ this is acknowledged
250 2.1.0 Sender OK

▷ We set the recipient (the real one, so that we really get the e-mail)
rcpt to: m.kohlhase@jacobs-university.de



▷ this is acknowledged
250 2.1.0 Recipient OK

▷ we tell the mail server that the mail data comes next
data

▷ this is acknowledged
354 Start mail input; end with <CRLF>.<CRLF>

▷ Now we can just type the a-mail, optionally with Subject, date,...
Subject: Test via SMTP
and now the mail body itself
.



▷ And a dot on a line by itself sends the e-mail off
250 2.6.0 <ed73c3f3-f876-4d03-98f2-e5ad5bbb6255@SHUBCAS04.jacobs.jacobs-university.de>
[InternalId=965770] Queued mail for delivery

 ©: Michael Kohlhase 389 

SMTP over telnet: Disconnecting

▷ That was almost all, but we close the connection (this is a telnet command)
quit

▷ our terminal server (the telnet program) tells us
221 2.0.0 Service closing transmission channel
Connection closed by foreign host.

 ©: Michael Kohlhase 390 

Essentially, the SMTP protocol mimics a conversation of polite computers that exchange messages by reading them out loud to each other (including the addressing information).

We could go on for quite a while with understanding one Internet protocol after each other, but this is beyond the scope of this course (indeed there are specific courses that do just that). Here we only answer the question where these protocols come from, and where we can find out more about them.

Internet Standardization

- ▷ **Question:** Where do all the protocols come from? (someone has to manage that)
- ▷ **Definition 15.2.24** The **Internet Engineering Task Force (IETF)** is an open standards organization that develops and standardizes Internet standards, in particular the TCP/IP and Internet protocol suite.
- ▷ All participants in the IETF are volunteers (usually paid by their employers)
- ▷ **Rough Consensus and Running Code:** Standards are determined by the “rough consensus method” (consensus preferred, but not all members need agree) IETF is interested in practical, working systems that can be quickly implemented.
- ▷ **Idea:** running code leads to rough consensus or vice versa.
- ▷ **Definition 15.2.25** The standards documents of the IETF are called **Request for Comments (RFC)**. (more than 6300 so far; see <http://www.rfc-editor.org/>)



This concludes our very brief exposition of the Internet. The essential idea is that it consists of a decentrally managed, packet-switched network whose function and value is defined in terms of the Internet protocol suite.

15.3 Basic Concepts of the World Wide Web

The World Wide Web (WWW) is the hypertext/multimedia part of the Internet. It is implemented as a service on top of the Internet (at the application level) based on specific protocols and markup formats for documents.

Concepts of the World Wide Web

- ▷ **Definition 15.3.1** A **web page** is a document on the WWW that can include multimedia data and hyperlinks.
- ▷ **Definition 15.3.2** A **web site** is a collection of related Web pages usually designed or controlled by the same individual or company.
- ▷ a web site generally shares a common domain name.
- ▷ **Definition 15.3.3** A **hyperlink** is a reference to data that can immediately be followed by the user or that is followed automatically by a user agent.
- ▷ **Definition 15.3.4** A collection text documents with hyperlinks that point to



Note that some forms of URIs can be used for actually locating (or accessing) the identified resources, e.g. for retrieval, if the resource is a document or sending to, if the resource is a mailbox. Such URIs are called “uniform resource *locators*”, all others “uniform resource *names*”.

Uniform Resource Names and Locators

- ▷ **Definition 15.3.9** A **uniform resource locator (URL)** is a URI that gives access to a web resource, by specifying an access method or location. All other URIs are called **uniform resource names (URN)**.
- ▷ **Idea:** A URN defines the identity of a resource, a URL provides a method for finding it.
- ▷ **Example 15.3.10** The following URI is a URL (try it in your browser)


```
http://kwarc.info/kohlhase/index.html
```
- ▷ **Example 15.3.11** `urn:isbn:978-3-540-37897-6` only identifies [Koh06] (it is in the library)
- ▷ **Example 15.3.12** URNs can be turned into URL via a catalog service, e.g. `http://wm-urn.org/urn:isbn:978-3-540-37897-6`
- ▷ **Note:** URI/URLs are one of the core features of the web infrastructure, they are considered to be the **plumbing of the WWW**. (direct the flow of data)



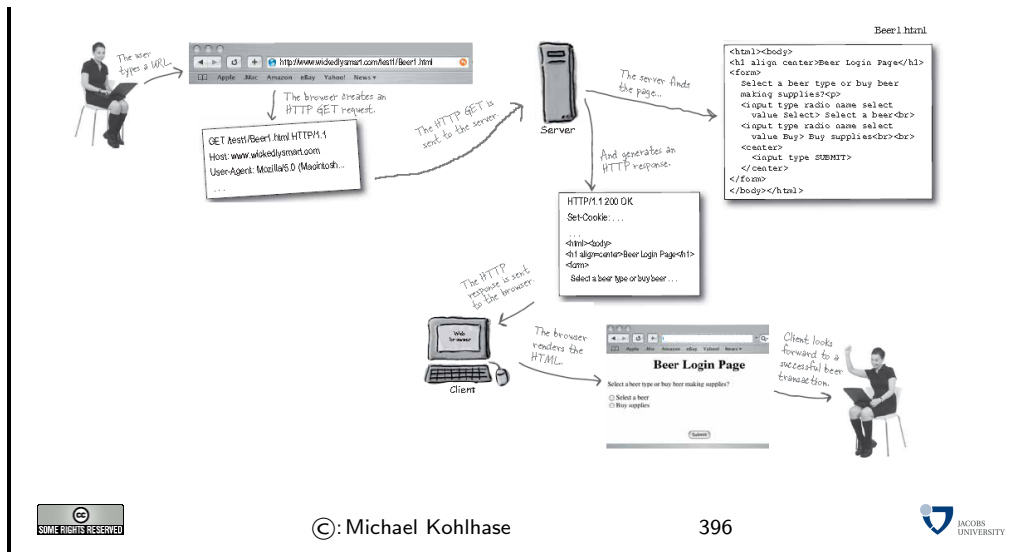
Historically, started out as URLs as short strings used for locating documents on the Internet. The generalization to identifiers (and the addition of URNs) as a concept only came about when the concepts evolved and the application layer of the Internet grew and needed more structure.

Note that there are two ways in URIs can fail to be resource locators: first, the scheme does not support direct access (as the ISBN scheme in our example), or the scheme specifies an access method, but address does not point to an actual resource that could be accessed. Of course, the problem of “dangling links” occurs everywhere we have addressing (and change), and so we will neglect it from our discussion. In practice, the URL/URN distinction is mainly driven by the scheme part of a URI, which specifies the access/identification scheme.

15.3.2 Running the World Wide Web

The infrastructure of the WWW relies on a client-server architecture, where the servers (called web servers) provide documents and the clients (usually web browsers) present the documents to the (human) users. Clients and servers communicate via the http protocol. We give an overview via a concrete example before we go into details.

The World Wide Web as a Client/Server System



We will now go through and introduce the infrastructure components of the WWW in the order we encounter them. We start with the user agent; in our example the web browser used by the user to request the web page by entering its URL into the URL bar.

Web Browsers

- ▷ **Definition 15.3.13** A **web Browser** is a software application for retrieving, presenting, and traversing information resources on the World Wide Web, enabling users to view Web pages and to jump from one page to another.
- ▷ **Practical Browser Tools:**
 - ▷ Status Bar: security info, page load progress
 - ▷ Favorites (bookmarks)
 - ▷ View Source: view the code of a Web page
 - ▷ Tools/Internet Options, history, temporary Internet files, home page, auto complete, security settings, programs, etc.
- ▷ **Example 15.3.14 (Common Browsers)**
 - ▷ MSInternetExplorer is provided by Microsoft for Windows (very common)
 - ▷ FireFox is an open source browser for all platforms, it is known for its standards compliance.
 - ▷ Safari is provided by Apple for MacOSX and Windows
 - ▷ Chrome is a lean and mean browser provided by Google
 - ▷ WebKit is a library that forms the open source basis for Safari and Chrome.

The web browser communicates with the web server through a specialized protocol, the hypertext transfer protocol, which we cover now.

HTTP: Hypertext Transfer Protocol

- ▷ **Definition 15.3.15** The **Hypertext Transfer Protocol** (HTTP) is an application layer protocol for distributed, collaborative, hypermedia information systems.
- ▷ June 1999: HTTP/1.1 is defined in RFC 2616 [FGM+99].

Definition 15.3.16 HTTP is used by a client (called **user agent**) to access web resources (addressed by Uniform Resource Locators (URLs)) via a **http request**. The **web server** answers by supplying the resource

- ▷ Most important HTTP requests (5 more less prominent)

GET	Requests a representation of the specified resource.	safe
PUT	Uploads a representation of the specified resource.	idempotent
DELETE	Deletes the specified resource.	idempotent
POST	Submits data to be processed (e.g., from a web form) to the identified resource.	

- ▷ **Definition 15.3.17** We call a HTTP request **safe**, iff it does not change the state in the web server. (except for server logs, counters, . . . ; no side effects)
- ▷ **Definition 15.3.18** We call a HTTP request **idempotent**, iff executing it twice has the same effect as executing it once.
- ▷ HTTP is a stateless protocol (very memory-efficient for the server.)



Finally, we come to the last component, the web server, which is responsible for providing the web page requested by the user.

Web Servers

- ▷ **Definition 15.3.19** A **web server** is a network program that delivers web pages and supplementary resources to and receives content from user agents via the hypertext transfer protocol.
- ▷ **Example 15.3.20 (Common Web Servers)**
 - ▷ apache is an open source web server that serves about 60% of the WWWeb.
 - ▷ IIS is a proprietary server provided by Microsoft.
 - ▷ nginx is a lightweight open source web server.
- ▷ Even though web servers are very complex software systems, they come pre-installed on most UNIX systems and can be downloaded for Windows [XAM].



Now that we have seen all the components we fortify our intuition of what actually goes down the net by tracing the http messages.

Example: An http request in real life

- ▷ Connect to the web server (port 80) (so that we can see what is happening)

```
telnet www.kwarc.info 80
```

- ▷ Send off the GET request

```
GET /teaching/GenCS2.html http/1.1
Host: www.kwarc.info
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.4)
Gecko/20100413 Firefox/3.6.4
```

- ▷ Response from the server

```
HTTP/1.1 200 OK
Date: Mon, 03 May 2010 06:48:36 GMT
Server: Apache/2.2.9 (Debian) DAV/2 SVN/1.5.1 mod_fastcgi/2.4.6 PHP/5.2.6-1+lenny8 with
Suhosin-Patch mod_python/3.3.1 Python/2.5.2 mod_ssl/2.2.9 OpenSSL/0.9.8g
Last-Modified: Sun, 02 May 2010 13:09:19 GMT
ETag: "1c78b-db1-4859c2f221dc0"
Accept-Ranges: bytes
Content-Length: 3505
Content-Type: text/html

<!--This file was generated by ws2html.xsl. Do NOT edit manually! -->
<html xmlns="http://www.w3.org/1999/xhtml"><head>...</head></html>
```



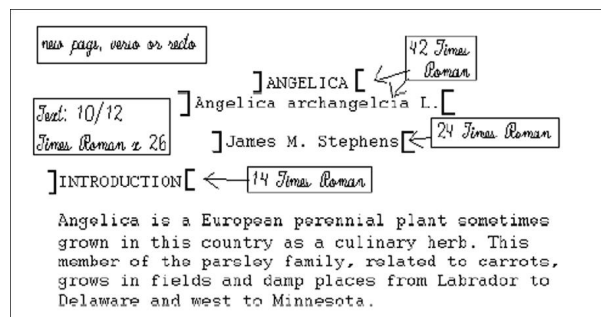
15.3.3 Multimedia Documents on the World Wide Web

We have seen the client-server infrastructure of the WWW, which essentially specifies how hypertext documents are retrieved. Now we look into the documents themselves.

In Section 5.2 we have already discussed how texts can be encoded in files. But for the rich documents we see on the WWW, we have to realize that documents are more than just sequences of characters. This is traditionally captured in the notion of document markup.

Document Markup

- ▷ **Definition 15.3.21 (Document Markup)** Document markup is the process of adding codes (special, standardized character sequences) to a document to control the structure, formatting, or the relationship among its parts.
- ▷ **Example 15.3.22** A text with markup codes (for printing)





There are many systems for document markup ranging from informal ones as in Definition 15.3.21 that specify the intended document appearance to humans – in this case the printer – to technical ones which can be understood by machines but serving the same purpose.

WWW documents have a specialized markup language that mixes markup for document structure with layout markup, hyper-references, and interaction. The HTML markup elements always concern text fragments, they can be nested but may not otherwise overlap. This essentially turns a text into a document tree.

HTML: Hypertext Markup Language

▷ **Definition 15.3.23** The **HyperText Markup Language** (HTML), is a representation format for web pages. Current version 4.01 is defined in [RHJ98].

▷ **Definition 15.3.24 (Main markup elements of HTML)** HTML marks up the structure and appearance of text with tags of the form `<e1>` (begin) and `</e1>` (end), where `e1` is one of the following

structure	html, head, body	metadata	title, link, meta
headings	h1, h2, ..., h6	paragraphs	p, br
lists	ul, ol, dl, ..., li	hyperlinks	a
images	img	tables	table, th, tr, td, ...
styling	style, div, span	old style	b, u, tt, i, ...
interaction	script	forms	form, input, button

▷ **Example 15.3.25** A (very simple) HTML file with a single paragraph.

```
<html>
  <body>
    <p>Hello GenCS students!</p>
  </body>
</html>
```



HTML was created in 1990 and standardized in version 4 in 1997. Since then there has HTML has been basically stable, even though the WWW has evolved considerably from a web of static web pages to a Web in which highly dynamic web pages become user interfaces for web-based applications and even mobile applets. Acknowledging the growing discrepancy, the W3C has started the standardization of version 5 of HTML.

HTML5: The Next Generation HTML

▷ **Definition 15.3.26** The **HyperText Markup Language** (HTML5), is believed to be the next generation of HTML. It is defined by the W3C and the WhatWG.

▷ HTML5 includes support for

- ▷ audio/video without plugins,
- ▷ a canvas element for scriptable, 2D, bitmapped graphics
- ▷ *SVG* for Scalable Vector Graphics
- ▷ MathML inline and display-style mathematical formulae

- ▷ The W3C is expected to issue a “recommendation” that standardizes HTML5 in 2014.
- ▷ Even though HTML5 is not formally standardized yet, almost all major web browsers already implement almost all of HTML5.



©: Michael Kohlhase

403



As the WWW evolved from a hypertext system purely aimed at human readers to an Web of multimedia documents, where machines perform added-value services like searching or aggregating, it became more important that machines could understand critical aspects web pages. One way to facilitate this is to separate markup that specifies the content and functionality from markup that specifies human-oriented layout and presentation (together called “styling”). This is what “cascading style sheets” set out to do. Another motivation for CSS is that we often want the styling of a web page to be customizable (e.g. for vision-impaired readers).

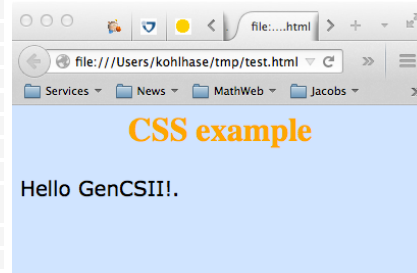
CSS: Cascading Style Sheets

- ▷ **Idea:** Separate structure/function from appearance.

Definition 15.3.27 The **Cascading Style Sheets** (CSS), is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents. Current version 2.1 is defined in [BCHL09].

- ▷ **Example 15.3.28** Our text file from Example 15.3.25 with embedded CSS

```
<html>
<head>a
  <style type="text/css">
    body {background-color:#d0e4fe;}
    h1 {color:orange;
        text-align:center;}
    p {font-family:"Verdana";
        font-size:20px;}
  </style>
</head>
<body>
  <h1>CSS example</h1>
  <p>Hello GenCSII!.</p>
</body>
</html>
```



©: Michael Kohlhase

404



With the technology described so far, we can basically build static web pages: hypertexts, where interaction is limited to navigation via hyperlinks: they are highlighted in the layout, and when we select them (usually by clicking), we navigate to the link target (to a new web page or a text fragment in the same page). With a few additions to the technology, web pages can be made much more interactive and versatile, up to a point, where they can function as interfaces to applications which run on a web server.

15.4 Web Applications

In this section we show how with a few additions to the basic WWW infrastructure introduced in Section 15.2, we can turn web pages into web-based applications that can be used without having to install additional software.

The first thing we need is a means to send information back to the web server, which can be used as input for the web application. Fortunately, this is already foreseen by the HTML format.

HTML Forms: Submitting Information to the Web Server

▷ **Example 15.4.1** Forms contain input fields and explanations.

```
<form name="input" action="html_form_submit.asp" method="get">
  Username: <input type="text" name="user" />
  <input type="submit" value="Submit" />
</form>
```

The result is a form with three elements: a text, an input field, and a submit button, that will trigger a HTTP GET request to the URL specified in the action attribute.

Username:



As the WWW is based on a client-server architecture, computation in web applications can be executed either on the client (the web browser) or the server (the web server). For both we have a special technology; we start with computation on the web server.

Server-Side Scripting: Programming Web Pages

▷ **Idea:** Why write HTML pages if we can also program them! (easy to do)

▷ **Definition 15.4.2** A **server-side scripting framework** is a web server extension that generates web pages upon HTTP GET requests.

▷ **Example 15.4.3** perl is a scripting language with good string manipulation facilities. perl CGI is an early server-side scripting framework based on this.

▷ Server-side scripting frameworks allow to make use of external resources (e.g. databases or data feeds) and computational services during web page generation.

▷ **Problem:** Most web page content is static (page head, text blocks, etc.) (and no HTML editing support in program editors)

▷ **Idea:** Embed program snippets into HTML pages. (only execute these, copy rest)

▷ **Definition 15.4.4** A **server-side scripting language** is a server side scripting framework where web pages are generated from HTML documents with embedded program fragments that are executed in context during web page generation.

▷ **Note:** No program code is left in the resulting web page after generation (important security concern)



To get a concrete intuition on the possibilities of server-side scripting frameworks, we will

present PHP, a commonly used open source scripting framework. There are many other examples, but they mainly differ on syntax and advanced features.

PHP, a Server-Side Scripting Language

▷ **Definition 15.4.5** PHP (originally “Programmable Home Page Tools”, later “PHP: Hypertext Processor”) is a server-side scripting language with a C-like syntax. PHP code is embedded into HTML via special “tags” `<?php` and `?>`

▷ **Example 15.4.6** The following PHP program uses `echo` for string output

```
<html>
  <body><?php echo 'Hello world';?></body>
</html>
```

▷ **Example 15.4.7** We can access the server clock in PHP (and manipulate it)

```
<?php
$tomorrow = mktime(0,0,0,date("m"),date("d")+1,date("Y"));
echo "Tomorrow is ".date("d. m. Y", $tomorrow);
?>
This fragment inserts tomorrow's date into a web page
```

▷ **Example 15.4.8** We can generate pages from a database (here MySQL)

```
<?php
$con = mysql_connect("localhost","peter","abc123");
if (!$con)
{
  die('Could not connect: ' . mysql_error());
}

mysql_select_db("my_db", $con);

$result = mysql_query("SELECT * FROM Persons");

while($row = mysql_fetch_array($result))
{
  echo $row['FirstName'] . " " . $row['LastName'];
  echo "<br />";
}

mysql_close($con);
?>
```

▷ **Example 15.4.9** We can even send e-mail via this e-mail form.

```
<html><body>
<?php
if (isset($_REQUEST['email'])){//if "email" is filled out, send email
  //send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail("someone@example.com", $subject,
  $message, "From:" . $email);
  echo "Thank you for using our mail form";}
else //if "email" is not filled out, display the form
{echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";}
?>
</body></html>
```



With server-side scripting frameworks like PHP, we can already build web applications, which we now define.

Web Applications: Using Applications without Installing

▷ **Definition 15.4.10** A **web application** is a website that serves as a user interface for a server-based application using a web browser as the client.

▷ **Example 15.4.11** Commonly used web applications include

- ▷ `http://ebay.com`; auction pages are generated from databases
- ▷ `http://www.weather.com`; weather information generated weather feeds
- ▷ `http://slashdot.org`; aggregation of news feeds/discussions
- ▷ `http://github.com`; source code hosting and project management

Common Traits: pages generated from databases and external feeds, content submission via HTML forms, file upload

▷ **Definition 15.4.12** A **web application framework** is a software framework for creating web applications.

▷ **Example 15.4.13** The LAMP stack is a web application framework based on linux, apache, MySQL, and PHP.

▷ **Example 15.4.14** A variant of the LAMP stack is available for Windows as XAMPP [XAM].



Indeed, the first web applications were essentially built in this way. Note however, that as we remarked above, no PHP code remains in the generated web pages, which thus “look like” static web pages to the client, even though they were generated dynamically on the server.

There is one problem however with web applications that is difficult to solve with the technologies so far. We want web applications to give the user a consistent user experience even though they are made up of multiple web pages. In a regular application we only want to login once and expect the application to remember e.g. our username and password over the course of the various interactions with the system. For web applications this poses a technical problem which we now discuss.

State in Web Applications and Cookies

▷ **Recall:** Web applications contain multiple pages, HTTP is a stateless protocol.

▷ **Problem:** how do we pass state between pages? (e.g. **username, password**)

▷ **Simple Solution:** Pass information along in query part of page URLs.

▷ **Example 15.4.15 (HTTP GET for Single Login)** Since we are generating pages we can generate augmented links

```
<a href="http://example.org/more.html?user=joe,pass=hideme">... more</a>
```

Problem: only works for limited amounts of information and for a single session

▷ **Other Solution:** Store state persistently on the client hard disk

▷ **Definition 15.4.16** A **cookie** is a text file stored on the client hard disk by the web browser. Web servers can request the browser to store and send cookies.

▷ **Note:** cookies are data not programs, they do not generate pop-ups or behave like viruses, but they can include your log-in name and browser preferences.

▷ **Note:** cookies can be convenient, but they can be used to gather information about you and your browsing habits.

▷ **Definition 15.4.17** **third party cookies** are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)



Note that that both solutions to the state problem are not ideal, for usernames and passwords the URL-based solution is particularly problematic, since HTTP transmits URLs in GET requests without encryption, and in our example passwords would be visible to anybody with a packet sniffer. Here cookies are little better as cookies, since they can be requested by any website you visit.

We now turn to client-side computation

One of the main advantages of moving documents from their traditional ink-on-paper form into an electronic form is that we can interact with them more directly. But there are many more interactions than just browsing hyperlinks we can think of: adding margin notes, looking up definitions or translations of particular words, or copy-and-pasting mathematical formulae into a computer algebra system. All of them (and many more) can be made, if we make documents programmable. For that we need three ingredients: *i*) a machine-accessible representation of the document structure, and *ii*) a program interpreter in the web browser, and *iii*) a way to send programs to the browser together with the documents. We will sketch the WWWeb solution to this in the following.

Dynamic HTML

▷ **Observation:** The nested, markup codes turn HTML documents into trees.

▷ **Definition 15.4.18** The **document object model** (DOM) is a data structure for the HTML document tree together with a standardized set of access methods.

▷ **Note:** All browsers implement the DOM and parse HTML documents into it; only then is the DOM rendered for the user.

▷ **Idea:** generate parts of the web page dynamically by manipulating the DOM.

▷ **Definition 15.4.19** JavaScript is an object-oriented scripting language mostly used to enable programmatic access to the DOM in a web browser.

▷ JavaScript is standardized by ECMA in [ECM09].

- ▷ **Example 15.4.20** We write the some text into a HTML document object (the document API)

```
<html>
<head>
  <script type="text/javascript">document.write("Dynamic HTML!");</script>
</head>
<body><!-- nothing here; will be added by the script later --></body>
</html>
```



Let us fortify our intuition about dynamic HTML by going into a more involved example.

Applications and useful tricks in Dynamic HTML

- ▷ **Example 15.4.21** hide document parts by setting CSS style attribs to `display:none`

```
<html>
<head>
  <style type="text/css">#dropper { display: none; }</style>
  <script language="JavaScript" type="text/javascript">
    window.onload = function toggleDiv(element){
      if (document.getElementById(element).style.display == 'none')
        {document.getElementById(element).style.display = 'block'}
      else if (document.getElementById(element).style.display == 'block')
        {document.getElementById(element).style.display = 'none'}}
  </script>
</head>
<body>
  <button onclick="toggleDiv('dropper')">...more </button>
  <div id="dropper"><p>Now you see it!</p></div>
</body>
</html>
```

Application: write “gmail” or “google docs” as JavaScript enhanced web applications. (client-side computation for immediate reaction)

- ▷ **Current Megatrend:** Computation in the “cloud”, browsers (or “apps”) as user interfaces



Current web applications include simple office software (word processors, online spreadsheets, and presentation tools), but can also include more advanced applications such as project management, computer-aided design, video editing and point-of-sale. These are only possible if we carefully balance the effects of server-side and client-side computation. The former is needed for computational resources and data persistence (data can be stored on the server) and the latter to keep personal information near the user and react to local context (e.g. screen size).

We have now seen the basic architecture and protocols of the World Wide Web. This covers basic interaction with web pages via browsing of links, as has been prevalent until around 1995. But this is not how we interact with the web nowadays; instead of browsing we use web search engines like Google or Yahoo, we will cover next how they work.

15.5 Introduction to Web Search

In this section we will present an overview over the basic mechanisms of web search engines. They are important to understanding the WWW, since they have replaced web portals as the entry points of the WWW.

Web Search Engines

- ▷ **Definition 15.5.1** A **web search engine** is a web application designed to search for information on the World Wide Web.
- ▷ **Definition 15.5.2 (Types of Search Engines)** We call a web search engine
 - ▷ **human-organized**, if documents are categorized by subject-area experts, (e.g. **Open Directory, About**)
 - ▷ **computer-created** Software spiders crawl the web for documents and categorize pages, (e.g. **Google**)
 - ▷ **hybrid** if it combines the two categories above
 - ▷ **metasearch** if it sends direct queries to multiple search engines and aggregates/clusters results, (e.g. **Copernic, Vivisimo, Mamma**)



©: Michael Kohlhase

412

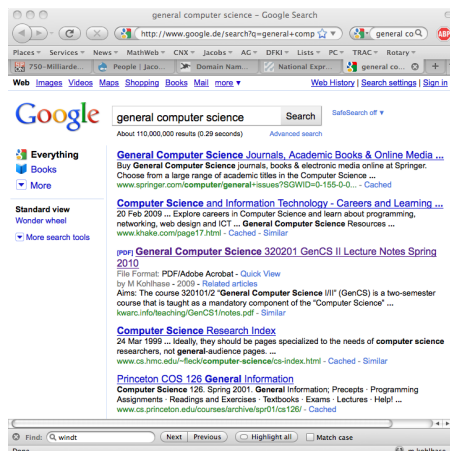


We will concentrate on computer-created search engines, since they are prevalent on the WWW nowadays. Let us see how they work.

Functional Schema of Web Search Engines

- ▷ Web search engines usually operate in four phases/components

- 1) **Data Acquisition**: a web crawler finds and retrieves (changed) web pages
- 2) **Search in Index**: write an index and search there.
- 3) **Sort the hits**: e.g. by importance
- 4) **Answer composition**: present the hits (and add advertisement)



©: Michael Kohlhase

413



We will now go through the four phases in turn and discuss specifics

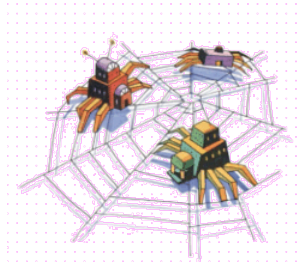
Data Acquisition for Web Search Engines: Web Crawlers

- ▷ **Definition 15.5.3** A **web crawler** or **spider** is a computer program that browses the WWW in an automated, orderly fashion for the purpose of information gathering.
- ▷ Web crawlers are mostly used for data acquisition of web search engines, but

can also automate web maintenance jobs (e.g. link checking).

- ▷ The WWW changes: 20% daily, 30% monthly, 50% never
- ▷ A Web crawler cycles over the following actions

- 1) reads web page
- 2) reports it home
- 3) finds hyperlinks
- 4) follows them



- ▷ **Note:** you can exclude web crawlers from your web site by configuring robots.txt.



Even though the image of a crawling insect suggests that, a web crawler is a program that lives on a host and stays there, downloading selected portions of the WWW. Actually, the picture we paint above is quite simplified, a web crawler has to be very careful not to download web pages multiple times to make progress. Recall that – seen as directed graphs – hypertexts may very well be cyclic. Additionally, much of the WWW content is only generated by web applications on user request, therefore modern web crawlers will try to generate queries that generate pages they can crawl.

The input for a web search engine is a query, i.e. a string that describes the set of documents to be referenced in the answer set. There are various types of query languages for information retrieval; we will only go into the most common ones here

Web Search: Queries

- ▷ **Definition 15.5.4** A **web search query** is a string that describes a set of document (fragments).
- ▷ **Example 15.5.5** Most web search engines accept **multiword queries**, i.e. multisets of strings (words).
- ▷ **Example 15.5.6** Many web search engines also accept **advanced query operators** and **wildcards**. (see below)
- ▷ **Definition 15.5.7** The following operators are called **wildcard s**.

?	(e.g. science? means search for the keyword “science” but I am not sure of the spelling)
*	(wildcard, e.g. comput* searches for keywords starting with comput combined with any word ending)

- ▷ **Definition 15.5.8** The following operators are called **advanced query operator s**

AND	(both terms must be present)
OR	(at least one of the terms must be present)

Also: Searches for various information formats & types, e.g. image search, scholarly search (require specialized query languages)



©: Michael Kohlhase

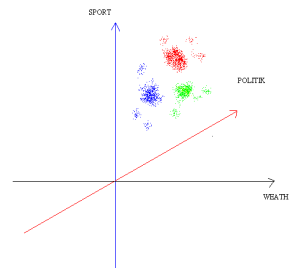
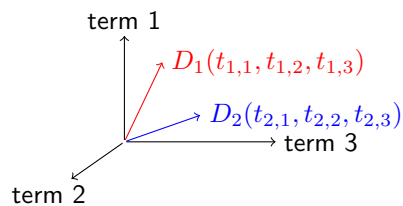
415



We now come to the central component of a web search engine: the indexing component. The main realization here is that with the size of the current web it is impossible to search the web linearly by comparing the query to the crawled documents one-by-one. So instead, the web search engine extracts salient features from the documents and stores them in a special data structure (usually tree-like) that can be queried instead of the documents themselves. In the prevalent information retrieval algorithms, the salient feature is a word frequency vector.

▷ Searching for Documents Efficiently: Indexing

- ▷ **Problem:** We cannot search the WWW linearly (even with 10^6 computers: $\geq 10^{15}B$)
- ▷ **Idea:** Write an “index” and search that instead. (like the index in a book)
- ▷ **Definition 15.5.9** Search engine **indexing** analyzes data and stores **key/data** pairs in a special data structure (the **search index** to facilitate efficient and accurate information retrieval.
- ▷ **Idea:** Use the words of a document as index (**multiword index**) The key for a document is the vector of word frequencies.



©: Michael Kohlhase

416



Note: The word frequency vectors used in the “vector space model” for information retrieval are very high-dimensional; the dimension is the number of words in the document corpus. Millions of dimensions are usual. However, linguistic methods like “stemming” (reducing words to word stems) are used to bring down the number of words in practice.

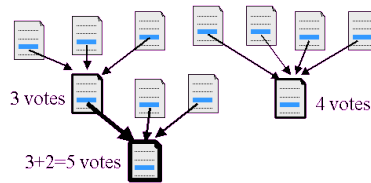
Once an answer set has been determined, the results have to be sorted, so that they can be presented to the user. As the user has a limited attention span – users will look at most at three to eight results before refining a query, it is important to rank the results, so that the hits that contain information relevant to the user’s information need early. This is a very difficult problem, as it involves guessing the intentions and information context of users, to which the search engine has no access.

Ranking Search Hits: e.g. Google’s Pagerank

- ▷ **Problem:** There are many hits, need to sort them by some criterion (e.g.

importance)

- ▷ **Idea:** A web site is important, ... if many other hyperlink to it.



- ▷ **Refinement:** ... , if many important web pages hyperlink to it.
- ▷ **Definition 15.5.10** Let A be a web page that is hyperlinked from web pages S_1, \dots, S_n , then

$$\text{PR}(A) = 1 - d + d \left(\frac{\text{PR}(S_1)}{C(S_1)} + \dots + \frac{\text{PR}(S_n)}{C(S_n)} \right)$$

where $C(W)$ is the number of links in a page W and $d = 0.85$.



Getting the ranking right is a determining factor for success of a search engine. In fact, the early of Google was based on the pagerank algorithm discussed above (and the fact that they figured out a revenue stream using text ads to monetize searches).

The final step for a web search engine is answer composition; at least, if the answer is addressed at a human user. The main task here is to assemble those information fragments that the user needs to determine whether the hit described contains information relevant to the respective information need.

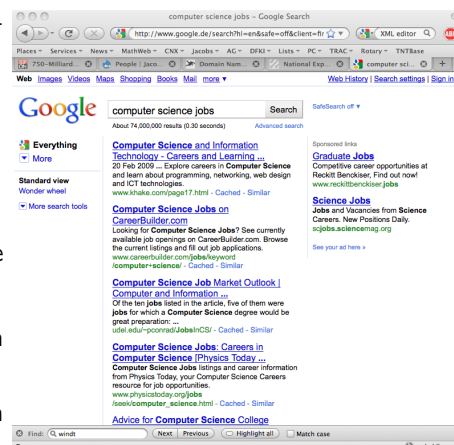
Answer Composition in Search Engines

- ▷ **Answers:** To present the search results we need to address:

- ▷ Hits and their context
- ▷ format conversion
- ▷ caching

- ▷ **Advertising:** to finance the service

- ▷ advertiser can buy search terms
- ▷ ads correspond to search interest
- ▷ advertiser pays by click.




Due to the gigantic size of the Internet, search engines are extremely resource-hungry web applications. The precise figures about the computational resources of the large internet companies are

well-kept trade secrets, but the following figure should give an intuition of the scales involved.

How to run

- ▷ **Google Hardware:** estimated 2003
 - ▷ 79,112 Computers (158,224 CPUs)
 - ▷ 316,448 Ghz computation power
 - ▷ 158,224 GB RAM
 - ▷ 6,180 TB Hard disk space
- ▷ 2010 Estimate: ~ 2 MegaCPU
- ▷ **Google Software:** Custom Linux Distribution



SOME RIGHTS RESERVED

©: Michael Kohlhase

419

JACOBS UNIVERSITY

So far we have completely neglected security issues on the Internet. They have become very important since the commercialization of the Internet in 1993 as we on the one hand nowadays use the internet for banking, trade, and confidential communication and on the other hand the Internet has become infested with malware, spyware, e-mail spam, and denial-of-service attacks. Fortunately, many of the underlying problems: authentication, authorization, and confidential communication can be solved by a joint technology: encryption.

15.6 Security by Encryption

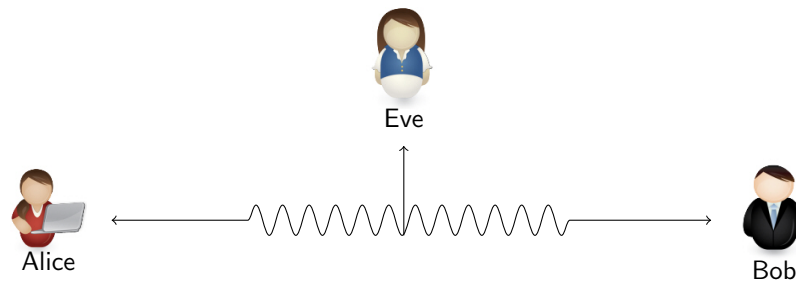
15.6.1 Introduction to Crypto-Systems

There are various ways to ensure security on networks: one is just to cease all traffic (not a very attractive one), another is to make the information on the network inaccessible by physical means (e.g. shielding the wires electrically and guarding them by a large police force). Here we want to look into “security by encryption”, which makes the content of Internet packets unreadable by unauthorized parties. We will start by reviewing the basics, and work our way towards a secure network infrastructure via the mathematical foundations and special protocols.

Security by Encryption

- ▷ **Problem:** In open packet-switched networks like the Internet, anyone
 - ▷ can inspect the packets (and see their contents via packet sniffers)
 - ▷ create arbitrary packets (and forge their metadata)
 - ▷ can combine both to falsify communication (man-in-the-middle attack)
- In “dedicated line networks” (e.g. old telephone) you needed switch room access.
- ▷ But there are situations where we want our communication to be confidential,

- ▷ Internet Banking (obviously, other criminals would like access to your account)
- ▷ Whistle-blowing (your employer should not know what you sent to WikiLeaks)
- ▷ Login to Campus.net (wouldn't you like to know my password to "correct" grades?)
- ▷ **The Situation:** Alice wants to communicate with Bob privately, but Eve (sdropper) can listen in



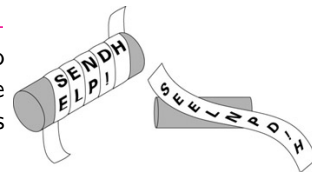
- ▷ **Idea:** Encrypt packet content (so that only the recipients can decrypt) and build this into the fabric of the Internet (so that users don't have to know)



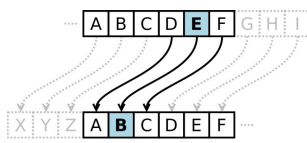
Encryption: Terminology & Examples

- ▷ **Definition 15.6.1** **Encryption** is the process of transforming information (referred to as **plaintext**) using an algorithm to make it unreadable to anyone except those possessing special knowledge, usually referred to as a **key**. The result of encryption is called **ciphertext**, and the reverse process that transforms ciphertext to plaintext: **decryption**. We call a method for encryption/decryption a **cipher**.
- ▷ **Definition 15.6.2** The corresponding science is called **cryptology**, it has two areas of study: **cryptography** (encryption/decryption via ciphers) and **code breaking/cryptoanalysis**: decrypting ciphertexts without a key or recovering keys from ciphertexts.
- ▷ **Example 15.6.3 (Spartan encryption (since ca. 700 BC))**

The oldest (military) encryption method is a **scytale**— a wooden stick of defined diameter, onto which a strip of parchment with letters can be wrapped to reveal the plaintext. Here the stick is the key and the parchment strip the ciphertext.



- ▷ **Example 15.6.4 (The Caesar Cipher)**



Shift the letters of the alphabet by n letters to the right. Julius Caesar (first mention) used 3, Augustus 1. Support by hardware.



▷ **Example 15.6.5 (Don't forget your Bank Card PIN)**

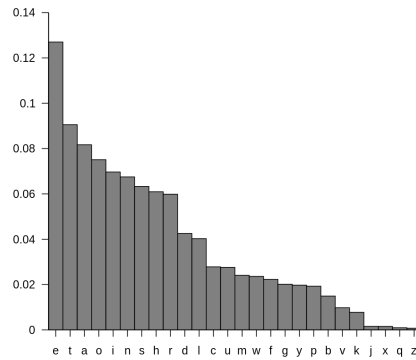


Write the encoded PIN number to the card, here complete each digit to 9. PIN = 5315



Code-Breaking, e.g. by Frequency-Analysis

- ▷ Letters (bigrams, trigrams,...) in English come in characteristic frequencies. (ETAOINSHRDLU)
- ▷ Use those to to decode a cipher text: most frequent character represents an "E", the second most frequent a "T", ...
- ▷ this works well for simple substitution ciphers



▷ **Data Paradox:** Deciphering longer texts is often easier than short ones.

▷ **Lesson for Encryption:** Change your cipher often (minimize data)



The simplest form of encryption (and the way we know from spy stories) uses uses the same key for encryption and decryption.

Symmetric Key Encryption

- ▷ **Definition 15.6.6 Symmetric-key cryptosystems** are a class of cryptographic algorithms that use essentially identical keys for both decryption and encryption.
- ▷ **Example 15.6.7** Permute the ASCII table by a bijective function $\varphi: \{0, \dots, 127\} \rightarrow \{0, \dots, 127\}$ (φ is the shared key)
- ▷ **Example 15.6.8** The AES algorithm (Advanced Encryption Standard) [AES01] is a widely used symmetric-key algorithm that is approved by US government organs for transmitting top-secret information. (efficient but safe)

- ▷ **AES is safe:** For AES-128/192/256, recovering the key takes $2^{126.1}/2^{189.7}/2^{254.4}$ steps respectively. (38/57/78 digit numbers)
- ▷ **Note:** For trusted communication sender and recipient need access to shared key.
- ▷ **Problem:** How to initiate safe communication over the internet? (far, far apart)
Need to exchange shared key (chicken and egg problem)
- ▷ **Pipe dream:** Wouldn't it be nice if I could just publish a key publicly and use that?
- ▷ **Actually:** this works, just (obviously) not with symmetric-key encryption.



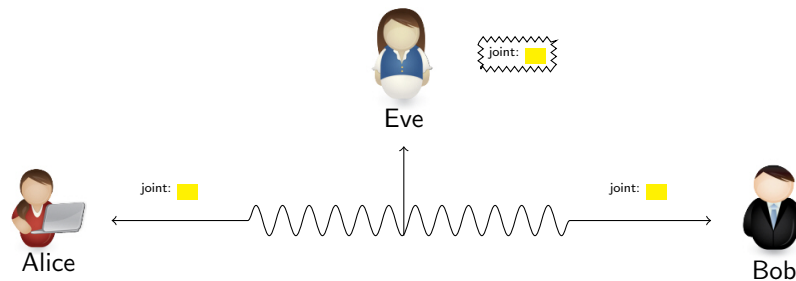
15.6.2 Public Key Encryption

To get around the chicken-and-egg problem of secure communication we identified above, we will introduce a more general way of encryption: one where we allow the keys for encryption and decryption to be different. This liberalization allows us to enter into a whole new realm of applications.

The following presentation is based on the one in [Con12]

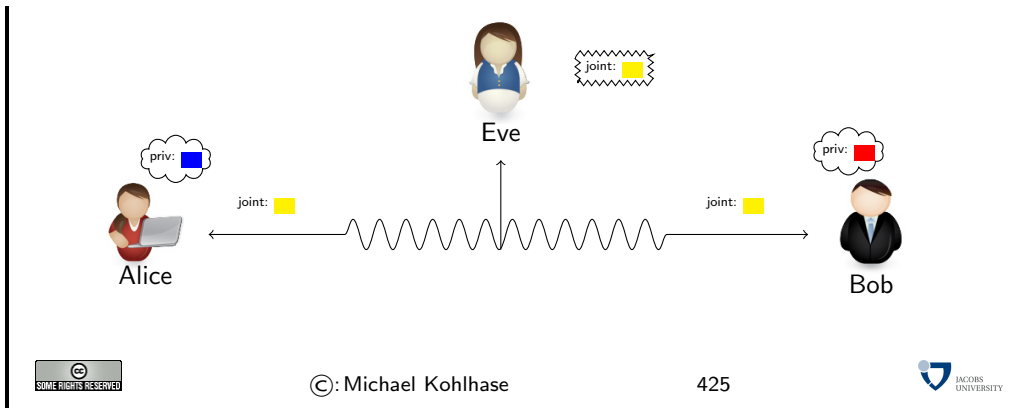
Diffie/Hellmann Key Exchange 1c

- ▷ Agree on a joint base color, here ■



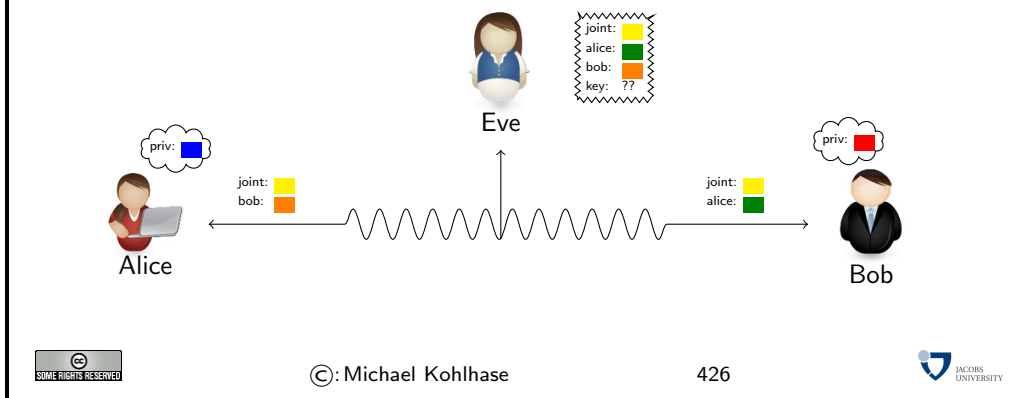
Diffie/Hellmann Key Exchange 2c

- ▷ randomly pick a private color (■/■).



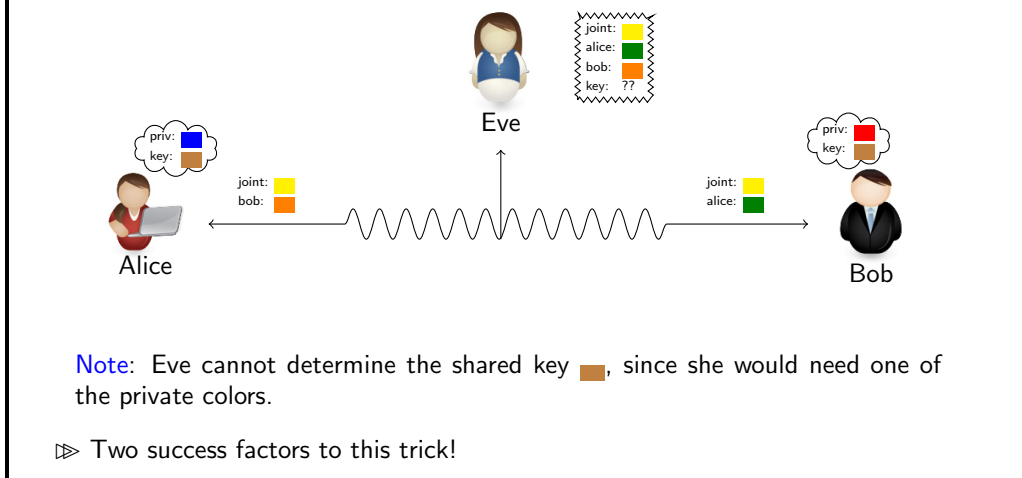
Diffie/Hellmann Key Exchange 3c

▷ mix private color into the joint base color [yellow square] to disguise it and send mixtures ([green square]/[orange square]) to the partner



Diffie/Hellmann Key Exchange 4c

▷ mix your own private color to get the key [brown square]



- ▷ mixing colors is associative and commutative (order/grouping irrelevant)
- ▷ mixing colors is much simpler than getting the original colors back from mixture



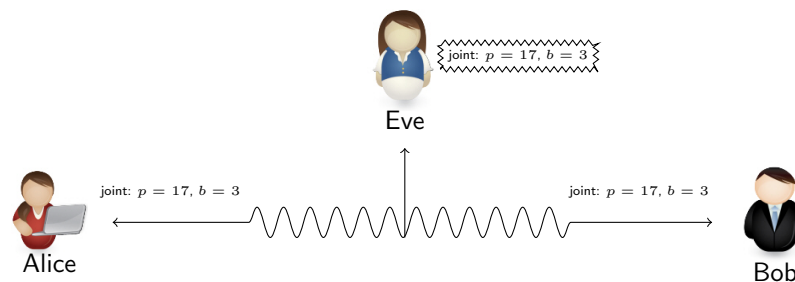
A numeric one-way function

- ▷ We need a one-way function for numbers to compute numeric keys.
- ▷ **Idea:** Take the discrete logarithm.
- ▷ **Definition 15.6.9 (Recap)** We say that a is congruent to b modulo m , iff $an = b$ and $0 \leq b < m$.
- ▷ **Idea:** We can do arithmetics modulo: $5 + 4 \equiv 2 \pmod{7}$ or $3^4 \equiv 1 \pmod{8}$.
- ▷ **Theorem 15.6.10 (A useful Fact)** If p is prime and b is a primitive root of n , then the $b^x \pmod{p}$ distribute evenly over $0 \leq x < p$.
- ▷ **Definition 15.6.11** Let p be a prime number, b a primitive root of n , and $b^x \equiv y \pmod{p}$ then we call x the discrete logarithm of y modulo p for the base k .
- ▷ **Observation 15.6.12** The discrete logarithm is very hard to compute: essentially p times the steps as for the discrete power. (generate and test)
- ▷ **Corollary 15.6.13** The discrete logarithm is a one-way function. (for large p)



Diffie/Hellmann Key Exchange 1

- ▷ Agree on a modulus m and a base b , e.g. $p = 17$, $b = 3$



Diffie/Hellmann Key Exchange 2

▷ randomly pick a private exponent ($e = 54/e = 24$), the private key.

Eve

Alice ← joint: $p = 17, b = 3$ → Bob

©: Michael Kohlhase 430 JACOBS UNIVERSITY

Diffie/Hellmann Key Exchange 3

▷ send $b^e \pmod p$ (the public key) to the partner ($3^{54} \equiv 15 \pmod{17}$ and $3^{24} \equiv 16 \pmod{17}$)

Eve

Alice ← joint: $p = 17, b = 3$, bob: 16 → Bob

Bob ← joint: $p = 17, b = 3$, alice: 15

©: Michael Kohlhase 431 JACOBS UNIVERSITY

Diffie/Hellmann Key Exchange 4

▷ raise the partner's public key to your private key

- ▷ Alice: $16^{54} \equiv 3^{24 \cdot 54} \equiv 3^{54 \cdot 24} \equiv 1 \pmod{17}$
- ▷ Bob: $15^{24} \equiv 3^{54 \cdot 24} \equiv 3^{24 \cdot 54} \equiv 1 \pmod{17}$

Note: Eve cannot determine the shared key 1, since she would need one of the private keys.

- ▷ Two success factors to this trick!
 - ▷ discrete exponentiation is associative and commutative (order/grouping irrelevant)
 - ▷ discrete logarithm is a one-way function.

©: Michael Kohlhase 432

Public Key Encryption

- ▷ **Definition 15.6.14** In an **asymmetric-key cryptosystem**, the key needed to encrypt a message is different from the key for decryption. Such a method is called a **public-key cryptosystem** if the the decryption key (the **private key**) is very difficult to reconstruct from encryption key (called the **public key**). We speak of a (cryptographic) **key pair**.
- ▷ Asymmetric cryptosystems are based on **trap door function** s: one-way functions that can (only) inverted with a suitable key.
- ▷ trap door functions are usually based on prime factorization.



©: Michael Kohlhase

433



Applications of Public-Key Kryptosystems

- ▷ **Preparation:** Create a cryptographic key pair and publishe the public key. (always keep the private key confidential!)
- ▷ **Application: Confidential Messaging:**

To send a confidential message the sender encrypts it using the intended recipient's public key; to decrypt the message, the recipient uses the private key.



- ▷ **Application: Digital Signatures:**

A **digital signature** consists of a plaintext together with its ciphertext – encoded with the sender's private key. A message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender had access to the private key (and therefore is likely to be the person associated with the public key used), and the part of the message that has not been tampered with.



The confidential messaging is analogous to a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender.

Note: For both applications (confidential messaging and digitally signed documents) we have only stated the basic idea. Technical realizations are more elaborate to be more efficient. One measure for instance is not to encrypt the whole message and compare the result of decrypting it, but only a well-chosen excerpt.

Let us now look at the mathematical foundations of encryption. It is all about the existence of natural-number functions with specific properties. Indeed cryptography has been a big and somewhat unexpected application of mathematical methods from number theory (which was perviously thought to be the ultimate pinnacle of “pure math”).

Encryption by Trapdoor Functions

- ▷ **Idea:** Mathematically, encryption can be seen as an injective function. Use functions for which the inverse (decryption) is difficult to compute.
- ▷ **Definition 15.6.15** A **one-way function** is a function that is “easy” to compute on every input, but “hard” to invert given the image of a random input.
- ▷ **In theory:** “easy” and “hard” are understood wrt. computational complexity theory, specifically the theory of polynomial time problems. E.g. “easy” $\hat{=}$ $O(n)$ and “hard” $\hat{=}$ $\Omega(2^n)$
- ▷ **Remark:** It is open whether one-way functions exist ($\hat{=}$ to $P = NP$ conjecture)
- ▷ **In practice:** “easy” is typically interpreted as “cheap enough for the legitimate users” and “prohibitively expensive for any malicious agents”.
- ▷ **Definition 15.6.16** A **trapdoor function** is a one-way function that is easy to invert given a piece of information called the **trapdoor**.
- ▷ **Example 15.6.17** Consider a padlock, it is easy to change from “open” to closed, but very difficult to change from “closed” to open unless you have a key (trapdoor).



Of course, we need to have one-way or trapdoor functions to get public key encryption to work. Fortunately, there are multiple candidates we can choose from. Which one eventually makes it into the algorithms depends on various details; any of them would work in principle.

Candidates for one-way/trapdoor functions

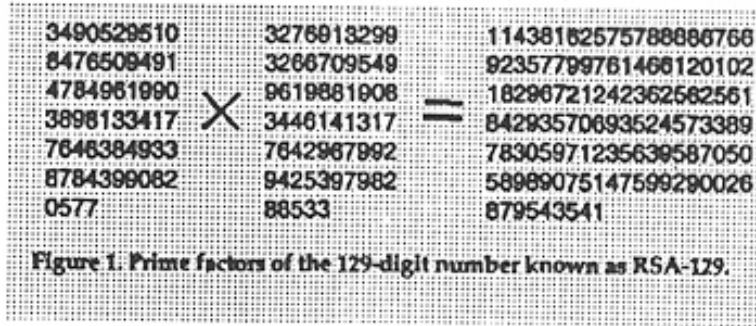
- ▷ **Multiplication and Factoring:** The function f takes as inputs two prime numbers p and q in binary notation and returns their product. This function can be computed in $O(n^2)$ time where n is the total length (number of digits) of the inputs. Inverting this function requires finding the factors of a given integer N . The best factoring algorithms known for this problem run in time $2^{O(\log(N)^{\frac{1}{3}} \log(\log(N))^{\frac{2}{3}})}$.
- ▷ **Modular squaring and square roots:** The function f takes two positive integers x and N , where N is the product of two primes p and q , and outputs $x^2 \bmod N$. Inverting this function requires computing square roots modulo N ; that is, given y and N , find some x such that $x^2 \bmod N = y$. It can be shown that the latter problem is computationally equivalent to factoring N (in the sense of polynomial-time reduction) (used in RSA encryption)
- ▷ **Discrete exponential and logarithm:** The function f takes a prime number p and an integer x between 0 and $p - 1$; and returns the $2^x \bmod p$. This discrete exponential function can be easily computed in time $O(n^3)$ where n is the number of bits in p . Inverting this function requires computing the discrete logarithm modulo p ; namely, given a prime p and an integer y between 0 and $p - 1$, find x such that $2^x = y$.



To see whether these trapdoor function candidates really behave as expected, RSA laboratories, one of the first security companies specializing in public key encryption has established a series of prime factorization challenges to test the assumptions underlying public key cryptography.

Example: RSA-129 problem

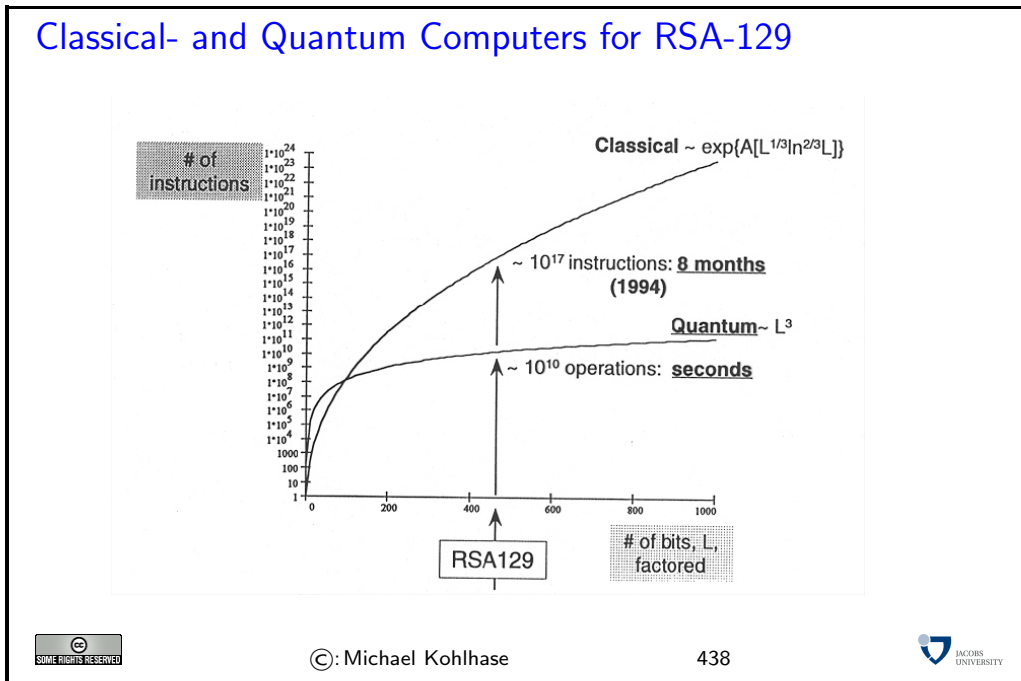
- ▷ **Definition 15.6.18** Call a number **semi-prime**, iff it has exactly two prime factors.
- ▷ These are exactly the numbers involved in RSA encryption.
- ▷ RSA laboratories initiated the RSA challenge, to see whether multiplication is indeed a “practical” trapdoor function
- ▷ **Example 15.6.19 (The RSA129 Challenge)** is to factor the semi-prime number on the right



- ▷ So far, the challenges up to ca 200 decimal digits have been factored, but all within the expected complexity bounds.
- ▷ **but:** would you report an algorithm that factors numbers in low complexity?



Note that all of these test are run on conventional hardware (von Neumann architectures); there have been claims that other computing hardware; most notably quantum computing or DNA computing might have completely different complexity theories, which might render these factorization problems tractable. Up to now, nobody has been able to actually build alternative computation hardware that can actually even attempt to solve such factorization problems (or they are not telling).



This concludes our excursion into theoretical aspects of encryption, we will now turn to the task of building these ideas into existing infrastructure of the Internet and the WWW. The most obvious thing we need to do is to publish public keys in a way that it can be verified to whom they belong.

15.6.3 Internet Security by Encryption

Public Key Certificates

- ▷ **Definition 15.6.20** A **public key certificate** is an electronic document which uses a digital signature to bind a public key with an identity, e.g. the name of a person or an organization.
- ▷ **Idea:** If we trust the signatory's signature, then we can use the certificate to verify that a public key belongs to an individual. Otherwise we verify the signature using the signatory's public key certificate.
- ▷ **Problem:** We can ascend the ladder of trust, but in the end we have to trust someone!
- ▷ In a typical public key infrastructure scheme, the signature will be of a **certificate authority**, an organization chartered to verify identity and issue public key certificates.
- ▷ In a "web of trust" scheme, the signature is of either the user (a self-signed certificate) or other users ("endorsements"). (e.g. PGP $\hat{=}$ Pretty Good Privacy)
- ▷ on a UNIX system, you can create a certificate (and associated private key) e.g. with (Windows similar \leadsto Google)


```
openssl ca -in req.pem -out newcert.pem
```



Building on the notion of a public key certificate, we can build secure variants of the application-level protocols. Of course, we could do this individually for every protocol, but this would duplicate efforts. A better way is to leverage the layered infrastructure of the Internet and build a generic secure transport-layer protocol, that can be utilized by all protocols that normally build on TCP or UDP.

Building Security in to the WWWeb Infrastructure

- ▷ **Idea:** Build Encryption into the WWWeb infrastructure (**make it easy to use**)
 \leadsto Secure variants of the application-level protocols that encrypt contents
- ▷ **Definition 15.6.21** **Transport layer security** (TLS) is a cryptographic protocol that encrypts the segments of network connections at the transport layer, using asymmetric cryptography for key exchange, symmetric encryption for privacy, and message authentication codes for message integrity.
- ▷ TLS can be used to make application-level protocols secure.



Let us now look at bit closer into the structure of the TLS handshake, the part of the TLS protocol that initiates encrypted communication.

A TLS Handshake between Client and Server

- ▷ **Definition 15.6.22** A **TLS handshake** authenticates a server and provides a shared key for symmetric-key encryption. It has the following steps
- 1) Client presents a list of supported encryption methods
 - 2) Server picks the strongest and tells client (C/S agree on method)
 - 3) Server sends back its public key certificate (name and public key)
 - 4) Client confirms certificate with CA (authenticates Server if successful)
 - 5) Client picks a random number, encrypts that (with servers public key) and sends it to server.
 - 6) Only server can decrypt it (using its private key)
 - 7) Now they both have a shared secret (the random number)
 - 8) From the random number, both parties generate key material
- ▷ **Definition 15.6.23** A **TLS connection** is a transport-layer connection secured by symmetric-key encryption. Authentication and keys are established by a TLS handshake and the connection is encrypted until it closes.



©: Michael Kohlhase

441



The reason we switch from public key to symmetric encryption after communication has been initiated and keys have been exchanged is that symmetric encryption is computationally more efficient without being intrinsically less secure.

But there is more to the integration of encryption into the WWW, than just enabling secure transport protocols. We need to extend the web servers and web browsers to implement the secure protocols (of course), and we need to set up a system of certification agencies, whose public keys are baked into web servers (so that they can check the signatures on public keys in server certificates). Moreover, we need user interfaces that allow users to inspect certificates, and grant exceptions, if needed.

Building Security in to the WWW Infrastructure

- ▷ **Definition 15.6.24 HTTP Secure** (HTTPS) is a variant of HTTP that uses TLS for transport. HTTPS URLs start with `https://`
- ▷ **Server Integration:** All common web servers support HTTPS on port 443 (default), but need a public key certificate. (self-sign one or buy one from a CA)
- ▷ **Browser Integration:** All common web browsers support HTTPS and give access to certificates



©: Michael Kohlhase

442



Confidential E-Mail with Digital Signatures

▷ **Hey:** That was a nice theoretical exercise, how can I use that in practice?

▷ **Example 15.6.25 (Secure E-Mail)**

Adding PGP (Pretty Good Privacy; an open source cryptosystem) to Thunderbird

- ▷ add the enigmail add-on to thunderbird (tools ↘ addons ↘ get addons)
- ▷ let it install GnuPG (the actual cryptosystem)
- ▷ let that generate a public/private key for your e-mail account (give password)

▷ Done! little sign/encrypt buttons appear on the lower right of your composition window.



Your mail visible to strangers

(here test mail to myself)

Note that the transport metadata on top are not encrypted

▷ Your mail after authentication

(here test mail to myself)

Note the verified signature shown on top.

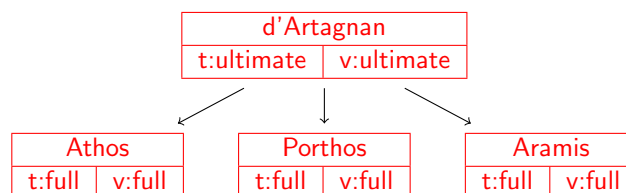
The Web of Trust

- ▷ **Recap:** We can only verify a signature, if we have, if we have a PKI certificate.
- ▷ **Cost Problem:** PKI certificates are expensive! (and authority needs to know you)
- ▷ **Centrality Problem:** What happens if a PKI authority has been compromised?
- ▷ **Idea:** instead of self-signed PKI certificates, mutually sign certificates in a “Web of Trust”
 - ▷ costs are minimal (we already know each other)
 - ▷ no central point of failure (more resilient)

Acknowledgement: The following presentation is adapted from [Rya]

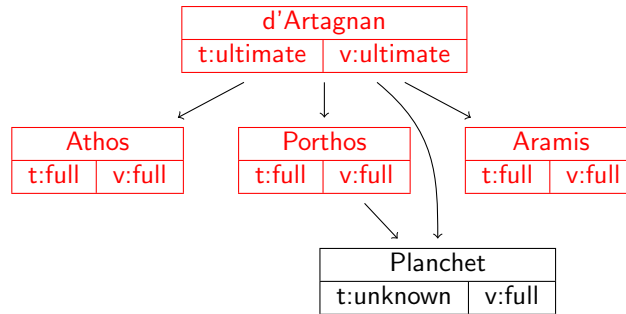
The Web of Trust for the Three Musketeers

- ▷ D'Artagnan arrives at Paris, and has a duel with Athos, Porthos, and Aramis, but they learn to trust each other against the guards of Cardinal Richilieu.
- ▷ To seal their friendship, they decide to exchange their public keys.
- ▷ **Definition 15.6.26** A key is called **valid**, iff it belongs to the individual it claims to belong to.
- ▷ To certify validity, the four friends also sign each-other's public key.
- ▷ **Example 15.6.27** d'Artagnan signs Porthos' key to say that *I, d'Artagnan, vouch that this key belongs to Porthos by adding my signature to it.*
- ▷ The musketeers also trust each other to make introductions.
- ▷ The situation from d'Artagnan's perspective



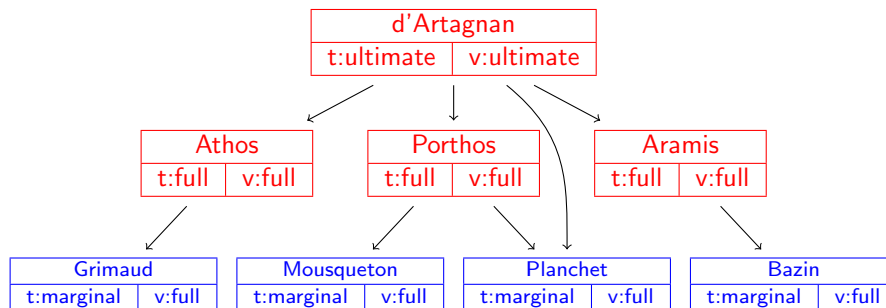
Porthos sends over Planchet as his new valet

- ▷ d'Artagnan can verify that Planchet is who he says he is, because his key bears Porthos' signature. (d'Artagnan has full trust in Porthos)
- ▷ d'Artagnan signs Planchet's key



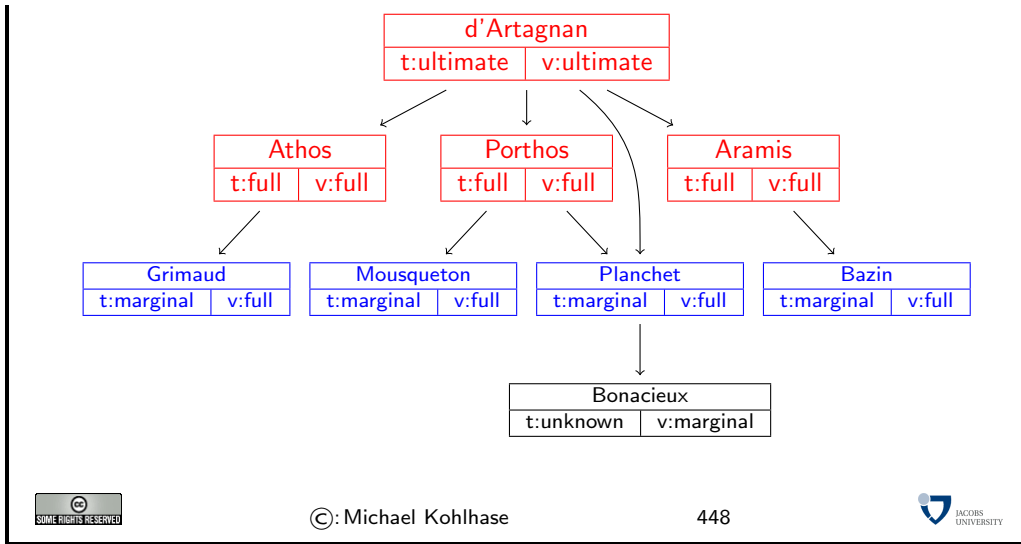
Setting (personal) Trust for other Keys

- ▷ Validity can be computed, but trust (in validity of introduced keys) must be set personally
- ▷ d'Artagnan sets the trust in all the valets to "marginal"



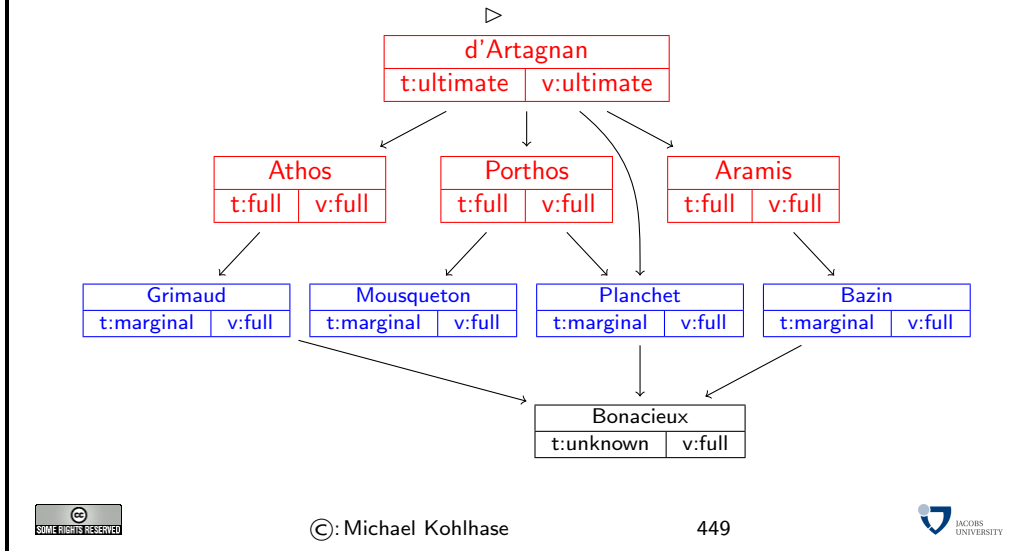
Understanding Marginal Trust

- ▷ **Rule 15.6.28** *No escalation of trust levels*
- ▷ Planchet introduces d'Artagnan to their landlord M. Bonacieux

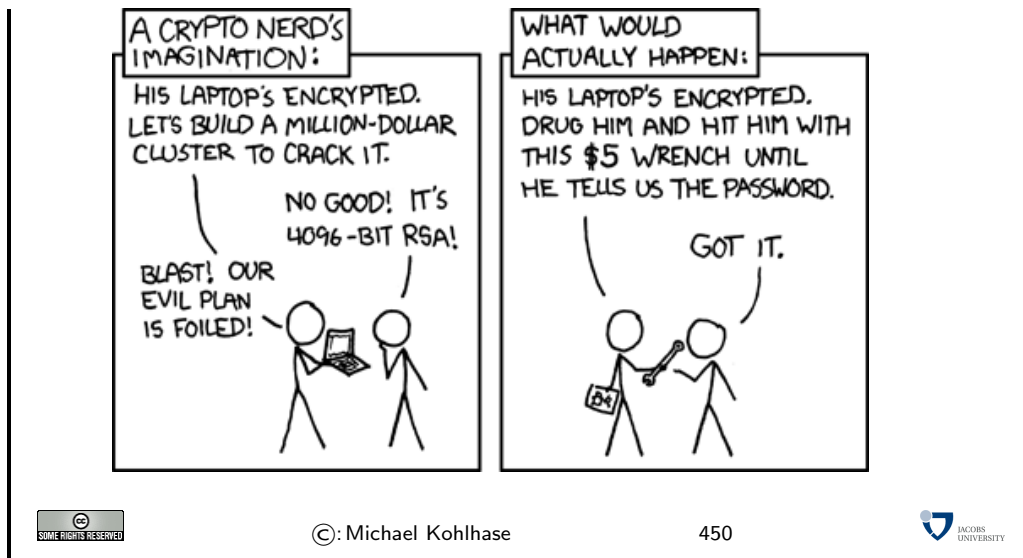


Understanding Marginal Trust

- ▷ All other valets also vouch for M. Bonacieux by signing his key
- ▷ Rule 15.6.29 $3 \text{ marginal} \hat{=} 1 \text{ full}$



But Technical Aspects of Security are not the only ones...



The next topic is an abstraction of the markup frameworks we have looked at so far. We introduce XML, the eXtensible Markup Language, which is used as a basis of many Internet and WWW technologies nowadays.

15.7 An Overview over XML Technologies

We have seen that many of the technologies that deal with marked-up documents utilize the tree-like structure of (the DOM) of HTML documents. Indeed, it is possible to abstract from the concrete vocabulary of HTML that the intended layout of hypertexts and the function of its fragments, and build a generic framework for document trees. This is what we will study in this section.

Excursion: XML (EXtensible Markup Language)

- ▷ XML is language family for the Web
 - ▷ tree representation language (begin/end brackets)
 - ▷ restrict instances by *Doc. Type Def. (DTD)* or *Schema* (Grammar)
 - ▷ Presentation markup by *style files* (XSL: XML Style Language)
- ▷ XML is extensible HTML & simplified SGML
- ▷ logic annotation (*markup*) instead of presentation!
- ▷ many tools available: parsers, compression, data bases, ...
- ▷ **conceptually**: transfer of directed graphs instead of strings.
- ▷ details at <http://www.w3c.org>

The idea of XML being an “extensible” markup language may be a bit of a misnomer. It is made “extensible” by giving language designers ways of specifying their own vocabularies. As such XML does not have a vocabulary of its own, so we could have also it an “empty” markup language that can be filled with a vocabulary.

XML is Everywhere (E.g. document metadata)

- ▷ **Example 15.7.1** Open a PDF file in AcrobatReader, then click on *File* \ DocumentProperties \ Document you get the following text: (showing only a small part)

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:ix='http://ns.adobe.com/ix/1.0/'>
  <rdf:Description xmlns:pdf='http://ns.adobe.com/pdf/1.3/'>
    <pdf:CreationDate>2004-09-08T16:14:07Z</pdf:CreationDate>
    <pdf:ModDate>2004-09-08T16:14:07Z</pdf:ModDate>
    <pdf:Producer>Acrobat Distiller 5.0 (Windows)</pdf:Producer>
    <pdf:Author>Herbert Jaeger</pdf:Author>
    <pdf:Creator>Acrobat PDFMaker 5.0 for Word</pdf:Creator>
    <pdf:Title>Exercises for ACS 1, Fall 2003</pdf:Title>
  </rdf:Description>
  ...
  <rdf:Description xmlns:dc='http://purl.org/dc/elements/1.1/'>
    <dc:creator>Herbert Jaeger</dc:creator>
    <dc:title>Exercises for ACS 1, Fall 2003</dc:title>
  </rdf:Description>
</rdf:RDF>
```



This is an excerpt from the document metadata which AcrobatDistiller saves along with each PDF document it creates. It contains various kinds of information about the creator of the document, its title, the software version used in creating it and much more. Document metadata is useful for libraries, bookselling companies, all kind of text databases, book search engines, and generally all institutions or persons or programs that wish to get an overview of some set of books, documents, texts. The important thing about this document metadata text is that it is not written in an arbitrary, PDF-proprietary format. Document metadata only make sense if these metadata are independent of the specific format of the text. The metadata that MSWord saves with each Word document should be in the same format as the metadata that Amazon saves with each of its book records, and again the same that the British library uses, etc.

XML is Everywhere (E.g. Web Pages)

- ▷ **Example 15.7.2** Open web page file in FireFox, then click on *View* \ PageSource, you get the following text: (showing only a small part and reformatting)

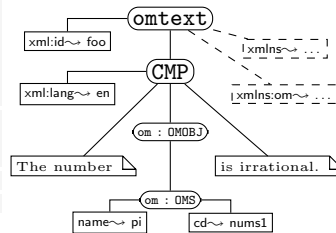
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Michael Kohlhase</title>
    <meta name="generator"
      content="Page generated from XML sources with the WSML package"/>
  </head>
  <body>...
  <p>
    <i>Professor of Computer Science</i><br/>
    Jacobs University<br/><br/>
    <strong>Mailing address - Jacobs (except Thursdays)</strong><br/>
    <a href="http://www.jacobs-university.de/schools/ses">
      School of Engineering & Science
    </a><br/>...
  </p>...
  </body>
</html>
```



XML Documents as Trees

- ▷ **Idea:** An XML Document is a Tree

```
<omtext xml:id="foo"
  xmlns="..."
  xmlns:om="...">
  <CMP xml:lang='en'>
    The number
    <om:OMOBJ>
      <om:OMS cd="nums1"
        name="pi"/>
    </om:OMOBJ>
    is irrational.
  </CMP>
</omtext>
```



- ▷ **Definition 15.7.3** The **XML document tree** is made up of **element nodes**, **attribute nodes**, **text nodes** (and **namespace declarations**, **comments**,...)
- ▷ **Definition 15.7.4** For communication this tree is serialized into a balanced bracketing structure, where
 - ▷ an element e_1 is represented by the brackets $\langle e_1 \rangle$ (called the **opening tag**) and $\langle /e_1 \rangle$ (called the **closing tag**).
 - ▷ The leaves of the tree are represented by **empty elements** (serialized as $\langle e_1 \rangle \langle /e_1 \rangle$, which can be abbreviated as $\langle e_1 / \rangle$)
 - ▷ and text nodes (serialized as a sequence of UniCode characters).
 - ▷ An element node can be annotated by further information using **attribute nodes** — serialized as an **attribute** in its opening tag

Note: As a document is a tree, the XML specification mandates that there must be a unique **document root**.



▷ The Document Object Model

- ▷ **Definition 15.7.5** The **document object model** (DOM) is a data structure for storing documents as marked-up documents as document trees together with a standardized set of access methods for manipulating them.



One of the great advantages of viewing marked-up documents as trees is that we can describe subsets of its nodes.

XPath, A Language for talking about XML Tree Fragments

- ▷ **Definition 15.7.6** The **XML path language** (XPath) is a language framework for specifying fragments of XML trees.
- ▷ **Example 15.7.7**

XPath exp.	fragment
/	root
omtext/CMP/*	all CMP children
//@name	the name attribute on the om : OMS element
//CMP/*[1]	the first child of all OMS elements
//*[@cd='nums1']	all elements whose cd has value nums1

©: Michael Kohlhase 456

An XPath processor is an application or library that reads an XML file into a DOM and given an XPath expression returns (pointers to) the set of nodes in the DOM that satisfy the expression. The final topic for our introduction to the information architecture of the Internet and the WWWeb is an excursion into the “Semantic Web”, a set of technologies that to make the WWWeb more machine-understandable.

15.8 The Semantic Web

The Semantic Web

- ▷ **Definition 15.8.1** The **semantic web** is a collaborative movement led by the W3C that promotes the inclusion of semantic content in web pages with the aim of converting the current web, dominated by unstructured and semi-structured documents into a machine-understandable “web of data”.
- ▷ **Idea:** Move web content up the ladder, use inference to make connections.

- ▷ **Example 15.8.2** We want to find information that is not explicitly represented (in one place)

Query: *Who was US president when Barak Obama was born?*
 Google: ... BIRTH DATE: August 04, 1961...

Query: *Who was US president in 1961?*
 Google: *President: Dwight D. Eisenhower [...] John F. Kennedy (starting January 20)*

Humans can read (and understand) the text and combine the information to get the answer.

The term “Semantic Web” was coined by Tim Berners Lee in analogy to semantic networks, only applied to the world wide web. And as for semantic networks, where we have inference processes that allow us the recover information that is not explicitly represented from the network (here the world-wide-web).

To see that problems have to be solved, to arrive at the “Semantic Web”, we will now look at a concrete example about the “semantics” in web pages. Here is one that looks typical enough.



What is the Information a User sees?

WWW2002
The eleventh International World Wide Web Conference
Sheraton Waikiki Hotel
Honolulu, Hawaii, USA
7-11 May 2002

Registered participants coming from
Australia, Canada, Chile Denmark, France, Germany, Ghana, Hong
Kong, India,
Ireland, Italy, Japan, Malta, New Zealand, The Netherlands, Nor-
way,
Singapore, Switzerland, the United Kingdom, the United States,
Vietnam, Zaire

On the 7th May Honolulu will provide the backdrop of the eleventh
International World Wide Web Conference.

Speakers confirmed
Tim Berners-Lee: Tim is the well known inventor of the Web,
Ian Foster: Ian is the pioneer of the Grid, the next generation
internet.


©: Michael Kohlhase
458


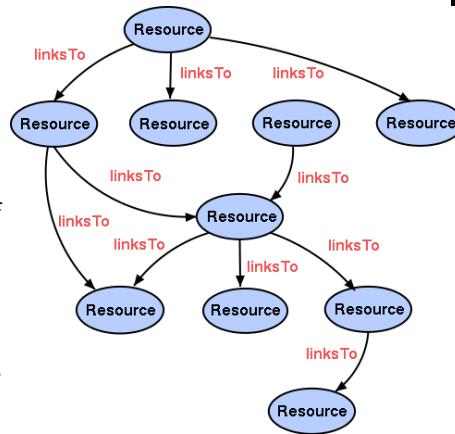
But as for semantic networks, what you as a human can see (“understand” really) is deceptive, so let us obfuscate the document to confuse your “semantic processor”. This gives an impression of what the computer “sees”.

What the machine sees

$WWW \in \mathbb{N}$
 $\mathcal{T}[\downarrow] \subseteq \mathbb{Z} \cup \mathbb{N} \cup \mathbb{R} \cup \mathbb{C} \cup \mathbb{H} \cup \mathbb{K}$
 $S(\mathbb{N} \cup \mathbb{R} \cup \mathbb{W} \cup \mathbb{H}) \cup \mathbb{H} \cup \mathbb{K}$
 $\mathcal{H}(\mathbb{N} \cup \mathbb{R} \cup \mathbb{H} \cup \mathbb{K}) \Leftrightarrow USA$
 $\mathbb{N} \cup \mathbb{R} \cup \mathbb{M} \cup \mathbb{H} \in \mathbb{N}$

$\mathcal{R} \cup \mathbb{N} \cup \mathbb{R} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{C} \cup \mathbb{D} \cup \mathbb{F} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{L} \cup \mathbb{M} \cup \mathbb{N} \cup \mathbb{O} \cup \mathbb{P} \cup \mathbb{Q} \cup \mathbb{R} \cup \mathbb{S} \cup \mathbb{T} \cup \mathbb{U} \cup \mathbb{V} \cup \mathbb{W} \cup \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$
 $\mathcal{A} \cup \mathbb{N} \cup \mathbb{R} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{C} \cup \mathbb{D} \cup \mathbb{F} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{L} \cup \mathbb{M} \cup \mathbb{N} \cup \mathbb{O} \cup \mathbb{P} \cup \mathbb{Q} \cup \mathbb{R} \cup \mathbb{S} \cup \mathbb{T} \cup \mathbb{U} \cup \mathbb{V} \cup \mathbb{W} \cup \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$
 $\mathcal{I} \cup \mathbb{N} \cup \mathbb{R} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{C} \cup \mathbb{D} \cup \mathbb{F} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{K} \cup \mathbb{L} \cup \mathbb{M} \cup \mathbb{N} \cup \mathbb{O} \cup \mathbb{P} \cup \mathbb{Q} \cup \mathbb{R} \cup \mathbb{S} \cup \mathbb{T} \cup \mathbb{U} \cup \mathbb{V} \cup \mathbb{W} \cup \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$

- ▷ **Resources:** identified by URI's, un-typed
- ▷ **Links:** href, src, ...limited, non-descriptive
- ▷ **User:** Exciting world - semantics of the resource, however, gleaned from content
- ▷ **Machine:** Very little information available - significance of the links only evident from the context around the anchor.



©: Michael Kohlhase

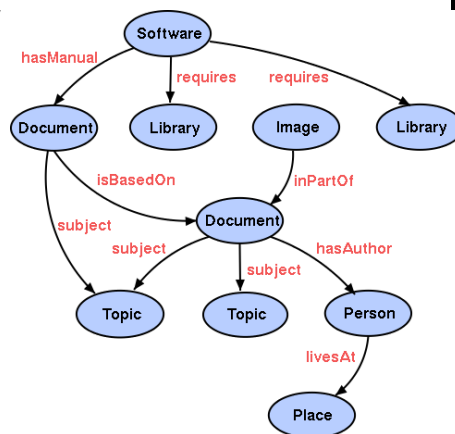
463



Let us now contrast this with the envisioned semantic web.

The Semantic Web

- ▷ **Resources:** Globally Identified by URI's or Locally scoped (Blank), Extensible, Relational
- ▷ **Links:** Identified by URI's, Extensible, Relational
- ▷ **User:** Even more exciting world, richer user experience
- ▷ **Machine:** More processable information is available (Data Web)
- ▷ **Computers and people:** Work, learn and exchange knowledge effectively



©: Michael Kohlhase

464



Essentially, to make the web more machine-processable, we need to classify the resources by the concepts they represent and give the links a meaning in a way, that we can do inference with that.

The ideas presented here gave rise to a set of technologies jointly called the “semantic web”, which we will now summarize before we return to our logical investigations of knowledge representation techniques.

Need to add “Semantics”

- ▷ External agreement on meaning of annotations E.g., Dublin Core
 - ▷ Agree on the meaning of a set of annotation tags
 - ▷ Problems with this approach: Inflexible, Limited number of things can be expressed

- ▷ Use Ontologies to specify meaning of annotations
 - ▷ Ontologies provide a vocabulary of terms
 - ▷ New terms can be formed by combining existing ones
 - ▷ Meaning (semantics) of such terms is formally specified
 - ▷ Can also specify relationships between terms in multiple ontologies
- ▷ Inference with annotations and ontologies (get out more than you put in!)
 - ▷ Standardize annotations in RDF [KC04] or RDFa [HASB13] and ontologies on OWL [OWL09]
 - ▷ Harvest RDF and RDFa in to a triplestore or OWL reasoner.
 - ▷ Query that for implied knowledge (e.g. chaining multiple facts from Wikipedia)
SPARQL: Who was US President when Barack Obama was Born?
DBPedia: John F. Kennedy (was president in August 1961)



Part IV

Search and Declarative Computation

In this part, we will take a look at two particular topics in computation.

The first is a class of algorithms that are generally (and thus often as a last resort, where other algorithms are missing) applicable to a wide class of problems that can be represented in a certain form.

The second applies what we have learnt to a new programming paradigm: after we have looked at functional programming in Chapter 3 and imperative languages in Section 14.0 and Section 14.2, we will combine the representation via logic (see Chapter 6) and backtracking search into a new programming paradigm: “logic programming”, or more abstractly put “declarative programming”.

ProLog is a simple logic programming language that exemplifies the ideas we want to discuss quite nicely. We will not introduce the language formally, but in concrete examples as we explain the theoretical concepts. For a complete reference, please consult the online book by Blackburn & Bos & Striegnitz <http://www.coli.uni-sb.de/~kris/learn-prolog-now/>.

Chapter 16

Problem Solving and Search

In this chapter, we will look at a class of algorithms called search algorithms. These are algorithms that help in quite general situations, where there is a precisely described problem, that needs to be solved.

16.1 Problem Solving

Before we come to the search algorithms themselves, we need to get a grip on the types of problems themselves and how we can represent them, and on what the various types entail for the problem solving process.

The first step is to classify the problem solving process by the amount of knowledge we have available. It makes a difference, whether we know all the factors involved in the problem before we actually are in the situation. In this case, we can solve the problem in the abstract, i.e. make a plan before we actually enter the situation (i.e. offline), and then when the problem arises, only execute the plan. If we do not have complete knowledge, then we can only make partial plans, and have to be in the situation to obtain new knowledge (e.g. by observing the effects of our actions or the actions of others). As this is much more difficult we will restrict ourselves to offline problem solving.

Problem solving

- ▷ **Problem:** Find algorithms that help solving problems in general
- ▷ **Idea:** If we can describe/represent problems in a standardized way, we may have a chance to find general algorithms.
We will use the following two concepts to describe problems
 - States** A set of possible situations in in our problem domain
 - Actions** A set of possible actions that get us from one state to another.Using these, we can view a sequence of actions as a solution, if it brings us into a situation, where the problem is solved.
- ▷ **Definition 16.1.1 Offline problem solving:** Acting only with complete knowledge of problem and solution
- ▷ **Definition 16.1.2 Online problem solving:** Acting without complete knowledge

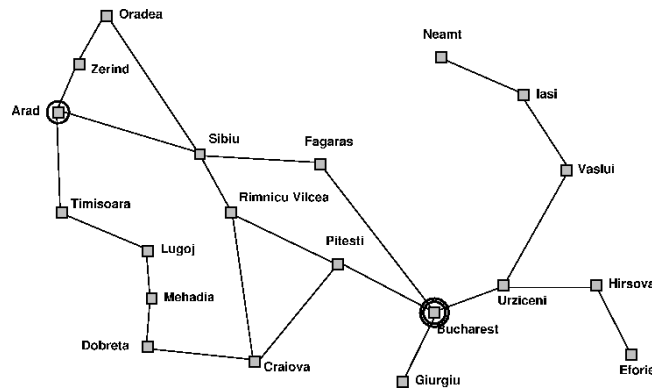
- ▷ Here: we are concerned with **offline** problem solving only.



We will use the following problem as a running example. It is simple enough to fit on one slide and complex enough to show the relevant features of the problem solving algorithms we want to talk about.

Example: Traveling in Romania

- ▷ **Scenario:** On holiday in Romania; currently in Arad, Flight leaves tomorrow from Bucharest.
- ▷ **Formulate problem:** *States:* various cities *Actions:* drive between cities
- ▷ **Solution:** Appropriate sequence of cities, e.g.: Arad, Sibiu, Fagaras, Bucharest



Given this example to fortify our intuitions, we can now turn to the formal definition of problem formulation and their solutions.

Problem Formulation

- ▷ The problem formulation models the situation at an appropriate level of abstraction. (do not model things like “put on my left sock”, etc.)
 - ▷ it describes the initial state (we are in Arad)
 - ▷ it also limits the objectives. (excludes, e.g. to stay another couple of weeks.)
- ▷ Finding the right level of abstraction and the required (not more!) information is often the key to success.
- ▷ **Definition 16.1.3** A **problem (formulation)** $\mathcal{P} := \langle S, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set S of **state** s and a set \mathcal{O} of **operator** s that specify how states can be accessed from each other. Certain states in S are designated as **goal state** s ($\mathcal{G} \subseteq S$) and there is a unique **initial state** \mathcal{I} .

- ▷ **Definition 16.1.4** A **solution** for a problem \mathcal{P} consists of a sequence of actions that bring us from \mathcal{I} to a goal state.



©: Michael Kohlhase

468



Note that this definition is very general, it applies to many many problems. So we will try to characterize these by difficulty.

Problem types

▷ Single-state problem

- ▷ observable (at least the initial state)
- ▷ deterministic (i.e. the successor of each state is determined)
- ▷ static (states do not change other than by our own actions)
- ▷ discrete (a countable number of states)

Multiple-state problem:

- ▷ ▷ initial state not/partially observable (multiple initial states?)
- ▷ deterministic, static, discrete

Contingency problem:

- ▷ ▷ non-deterministic (solution can branch, depending on contingencies)
- ▷ unknown state space (like a baby, agent has to learn about states and actions)



©: Michael Kohlhase

469



We will explain these problem types with another example. The problem \mathcal{P} is very simple: We have a vacuum cleaner and two rooms. The vacuum cleaner is in one room at a time. The floor can be dirty or clean.

The possible states are determined by the position of the vacuum cleaner and the information, whether each room is dirty or not. Obviously, there are eight states: $\mathcal{S} = \{1, 2, 3, 4, 5, 6, 7, 8\}$ for simplicity.

The goal is to have both rooms clean, the vacuum cleaner can be anywhere. So the set \mathcal{G} of goal states is $\{7, 8\}$. In the single-state version of the problem, $[right, suck]$ shortest solution, but $[suck, right, suck]$ is also one. In the multiple-state version we have $[right(\{2, 4, 6, 8\}), suck(\{4, 8\}), left(\{3, 7\}), suck(\{7\})]$.

Example: vacuum-cleaner world

▷ **Single-state Problem:**

- ▷ Start in 5
- ▷ **Solution:** $[right, suck]$

▷ **Multiple-state Problem:**

- ▷ Start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
- ▷ **Solution:** $[right, suck, left, suck]$
 - $right \rightarrow \{2, 4, 6, 8\}$
 - $suck \rightarrow \{4, 8\}$
 - $left \rightarrow \{3, 7\}$
 - $suck \rightarrow \{7\}$

©: Michael Kohlhase 470

Example: vacuum-cleaner world (continued)

▷ **Contingency Problem:**

- ▷ Murphy's Law: $suck$ can dirty a clean carpet
- ▷ Local sensing: $dirty / notdirty$ at location only
- ▷ Start in: $\{1, 3\}$
- ▷ **Solution:** $[suck, right, suck]$
 - $suck \rightarrow \{5, 7\}$
 - $right \rightarrow \{6, 8\}$
 - $suck \rightarrow \{6, 8\}$

▷ **better:** $[suck, right, \text{if dirt then suck}]$ (decide whether in 6 or 8 using local sensing)

©: Michael Kohlhase 471

In the contingency version of \mathcal{P} a solution is the following: $[suck(\{5, 7\}), right \rightarrow (\{6, 8\}), suck \rightarrow (\{6, 8\})]$, $[suck(\{5, 7\})]$, etc. Of course, local sensing can help: narrow $\{6, 8\}$ to $\{6\}$ or $\{8\}$, if we are in the first, then suck.



Single-state problem formulation

▷ Defined by the following four items

- 1) **Initial state:** (e.g. *Arad*)
- 2) **Successor function S :** (e.g. $S(Arad) = \{ \langle goZer, Zerind \rangle, \langle goSib, Sibiu \rangle, \dots \}$)
- 3) **Goal test:** (e.g. $x = Bucharest$ (explicit test))
 $noDirt(x)$ (implicit test)

4) **Path cost (optional):** (e.g. sum of distances, number of operators executed, etc.)

▷ **Solution:** A sequence of operators leading from the initial state to a goal state

 ©: Michael Kohlhase 472 

“Path cost”: There may be more than one solution and we might want to have the “best” one in a certain sense.

Selecting a state space



▷ **Abstraction:** Real world is absurdly complex
State space must be abstracted for problem solving

▷ **(Abstract) state:** Set of real states

▷ **(Abstract) operator:** Complex combination of real actions

▷ **Example:** *Arad* → *Zerind* represents complex set of possible routes

▷ **(Abstract) solution:** Set of real paths that are solutions in the real world

 ©: Michael Kohlhase 473 

“State”: e.g., we don’t care about tourist attractions found in the cities along the way. But this is problem dependent. In a different problem it may well be appropriate to include such information in the notion of state.

“Realizability”: one could also say that the abstraction must be sound wrt. reality.

Example: The 8-puzzle



7	2	4
5		6
8	3	1

1	2	3
4	5	6
7	8	

Start State

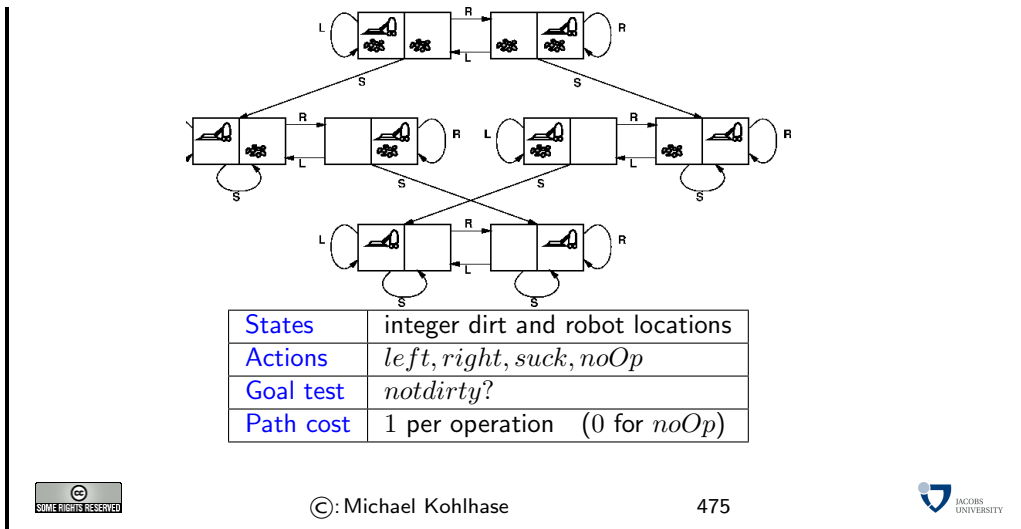
States	integer locations of tiles
Actions	<i>left, right, up, down</i>
Goal test	= goal state?
Path cost	1 per move

Goal State

 ©: Michael Kohlhase 474 

How many states are there? N factorial, so it is not obvious that the problem is in NP. One needs to show, for example, that polynomial length solutions do always exist. Can be done by combinatorial arguments on state space graph (really ?).

Example: Vacuum-cleaner

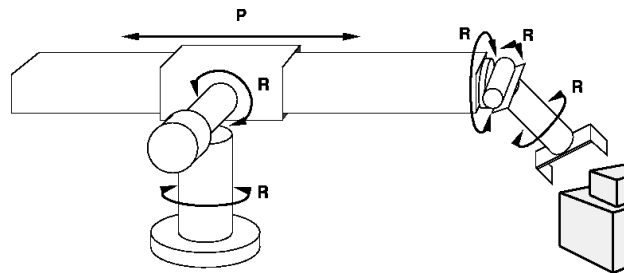


©: Michael Kohlhase

475



Example: Robotic assembly



States	real-valued coordinates of robot joint angles and parts of the object to be assembled
Actions	continuous motions of robot joints
Goal test	assembly complete?
Path cost	time to execute



©: Michael Kohlhase

476



16.2 Search

Tree search algorithms

- ▷ **Definition 16.2.1** The **tree search algorithm** consists of the simulated exploration of state space in a search tree by generating successors of already-explored states (Offline Algorithm)

```

procedure Tree-Search (problem, strategy) : <a solution or failure>
  <initialize the search tree using the initial state of problem>
  loop
    if <there are no candidates for expansion> <return failure> end if
    <choose a leaf node for expansion according to strategy>
    if <the node contains a goal state> return <the corresponding solution>
  
```



```

else <expand the node and add the resulting nodes to the search tree>
end if
end loop
end procedure
    
```

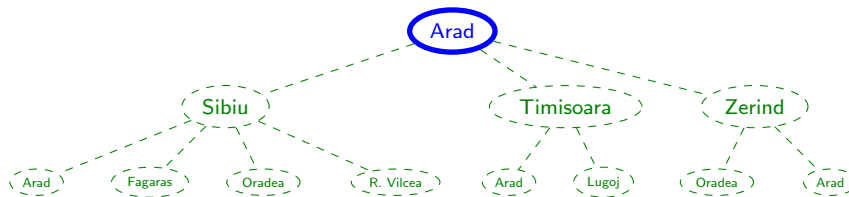


©: Michael Kohlhase

477



Tree Search: Example

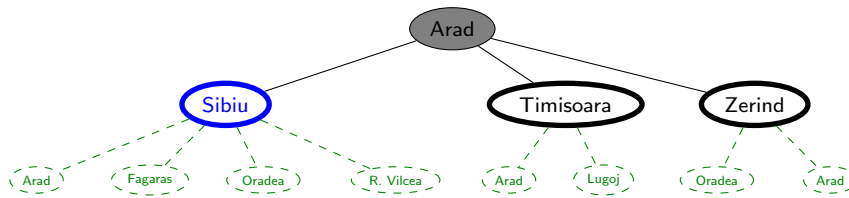


©: Michael Kohlhase

478



Tree Search: Example

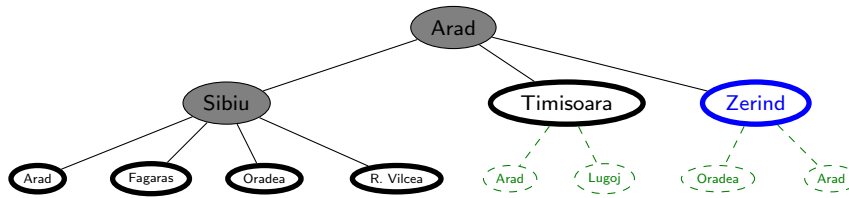


©: Michael Kohlhase

479



Tree Search: Example

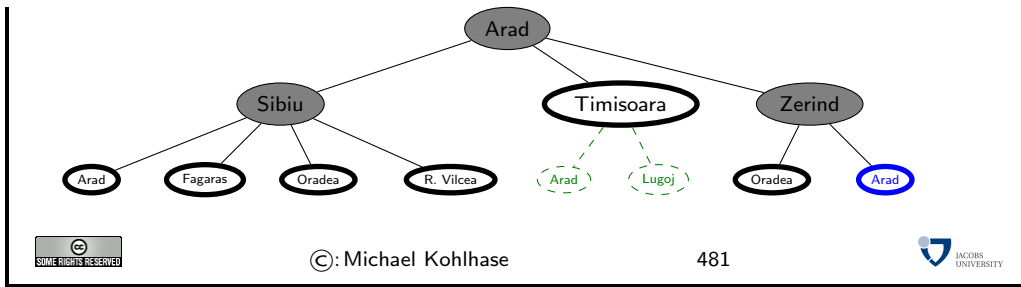


©: Michael Kohlhase

480

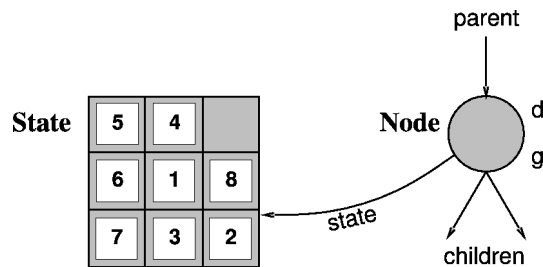


Tree Search: Example



Implementation: States vs. nodes

- ▷ A (representation of) a physical configuration
- ▷ A data structure constituting part of a search tree (includes *parent*, *children*, *depth*, *path cost*, etc.)



Implementation of search algorithms

```

procedure Tree_Search (problem, strategy)
  fringe := insert(make_node(initial_state(problem)))
  loop
    if fringe <is empty> fail end if
    node := first(fringe, strategy)
    if NodeTest(State(node)) return State(node)
    else fringe := insert_all(expand(node, problem), strategy)
    end if
  end loop
end procedure

```

- ▷ **Definition 16.2.2** The **fringe** is a list nodes not yet considered. It is ordered by the **search strategy** (see below)



STATE gives the state that is represented by *node*

EXPAND = creates new nodes by applying possible actions to *node*

A node is a data structure representing states, will be explained in a moment.

MAKE-QUEUE creates a queue with the given elements.

fringe holds the queue of nodes not yet considered.

REMOVE-FIRST returns first element of queue and as a side effect removes it from *fringe*.

STATE gives the state that is represented by *node*.

EXPAND applies all operators of the problem to the current node and yields a set of new nodes. INSERT inserts an element into the current *fringe* queue. This can change the behavior of the search.

INSERT-ALL Perform INSERT on set of elements.



Search strategies

- ▷ **Strategy:** Defines the **order** of node expansion
- ▷ **Important properties of strategies:**

completeness	does it always find a solution if one exists?
time complexity	number of nodes generated/expanded
space complexity	maximum number of nodes in memory
optimality	does it always find a least-cost solution?

- ▷ **Time and space complexity measured in terms of:**

<i>b</i>	maximum branching factor of the search tree
<i>d</i>	depth of a solution with minimal distance to root
<i>m</i>	maximum depth of the state space (may be ∞)


©: Michael Kohlhase
484




Complexity means here always *worst-case* complexity.

Note that there can be infinite branches, see the search tree for Romania.

16.3 Uninformed Search Strategies

Uninformed search strategies

- ▷ **Definition 16.3.1 (Uninformed search)** Use only the information available in the problem definition
- ▷ **Frequently used strategies:**
 - ▷ Breadth-first search
 - ▷ Uniform-cost search
 - ▷ Depth-first search
 - ▷ Depth-limited search
 - ▷ Iterative deepening search

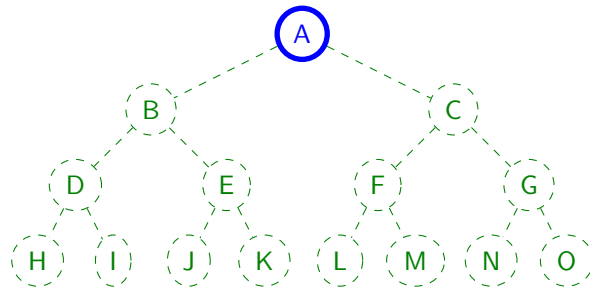

©: Michael Kohlhase
485


The opposite of uninformed search is informed or *heuristic* search. In the example, one could add, for instance, to prefer cities that lie in the general direction of the goal (here SE).

Uninformed search is important, because many problems do not allow to extract good heuristics.

Breadth-first search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



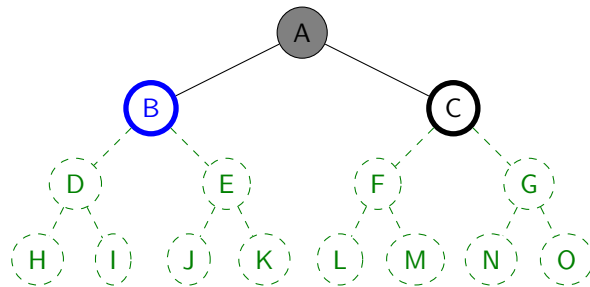
©: Michael Kohlhase

486



Breadth-First Search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



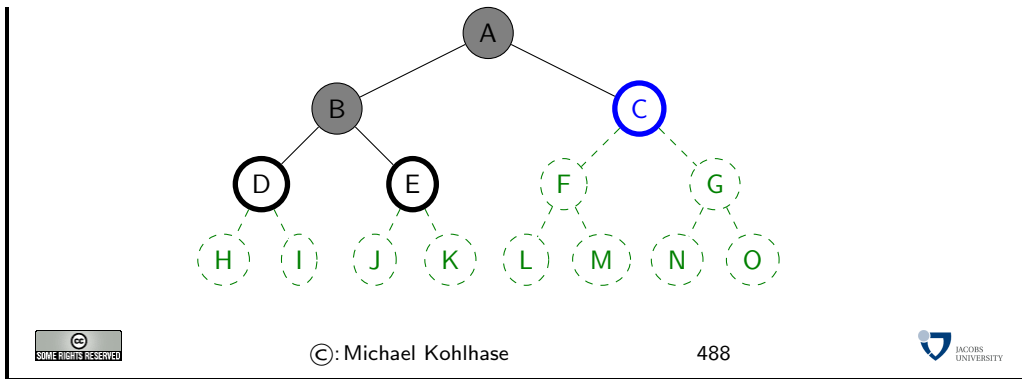
©: Michael Kohlhase

487



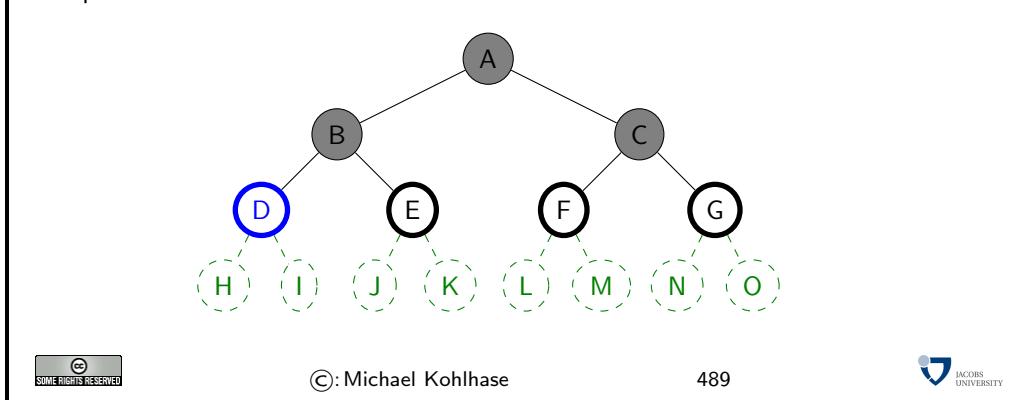
Breadth-First Search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



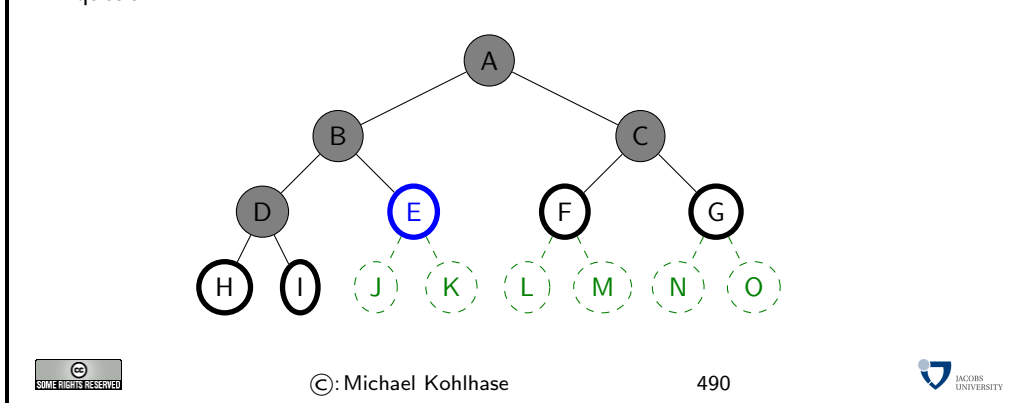
Breadth-First Search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue



Breadth-First Search

- ▷ **Idea:** Expand shallowest unexpanded node
- ▷ **Implementation:** *fringe* is a FIFO queue, i.e. successors go in at the end of the queue

©: Michael Kohlhase 491

We will now apply the breadth-first search strategy to our running example: Traveling in Romania. Note that we leave out the green dashed nodes that allow us a preview over what the search tree will look like (if expanded). This gives a much

Breadth-First Search: Romania

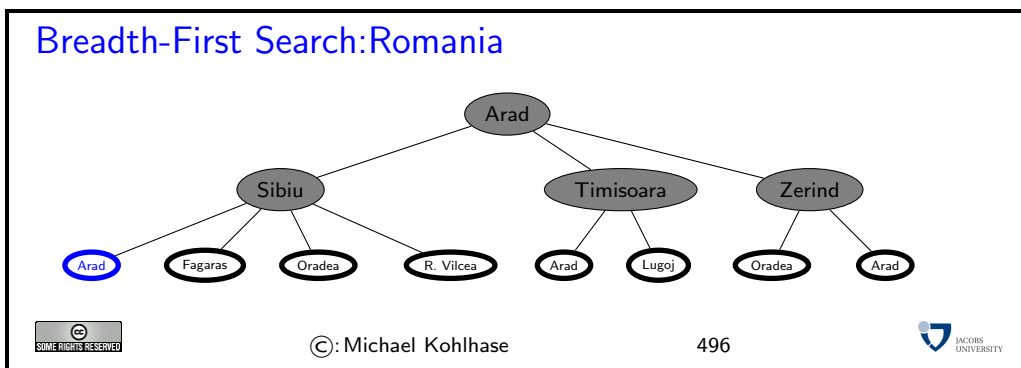
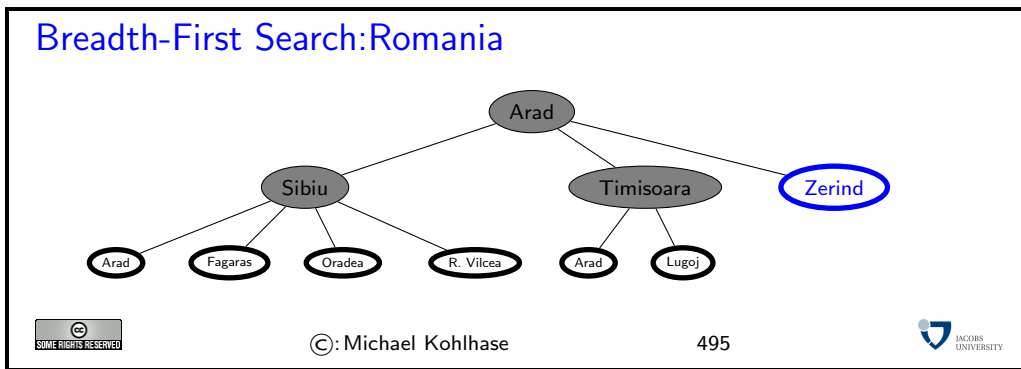
©: Michael Kohlhase 492

Breadth-First Search: Romania

©: Michael Kohlhase 493

Breadth-First Search: Romania

©: Michael Kohlhase 494



Breadth-first search: Properties

Complete	Yes (if b is finite)
Time	$1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) \in O(b^{d+1})$ i.e. exponential in d
Space	$O(b^{d+1})$ (keeps every node in memory)
Optimal	Yes (if cost = 1 per step), not optimal in general

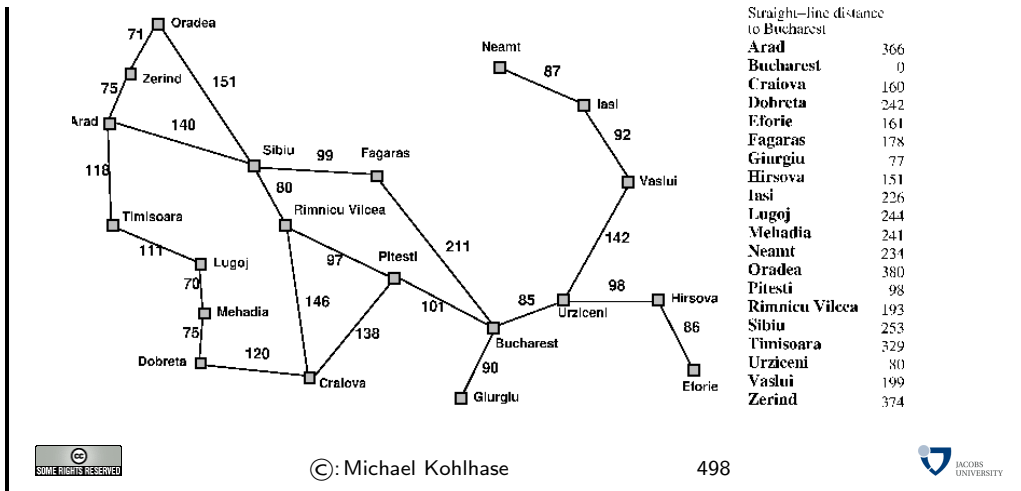
- ▷ **Disadvantage:** Space is the big problem (can easily generate nodes at 5MB/sec so 24hrs = 430GB)
- ▷ **Optimal?:** if cost varies for different steps, there might be better solutions below the level of the first solution.
- ▷ An alternative is to generate *all* solutions and then pick an optimal one. This works only, if m is finite.

©: Michael Kohlhase 497

The next idea is to let cost drive the search. For this, we will need a non-trivial cost function: we will take the distance between cities, since this is very natural. Alternatives would be the driving time, train ticket cost, or the number of tourist attractions along the way.

Of course we need to update our problem formulation with the necessary information.

Romania with Step Costs as Distances



©:Michael Kohlhase

498



Uniform-cost search

- ▷ **Idea:** Expand least-cost unexpanded node
- ▷ **Implementation:** fringe is queue ordered by increasing path cost.
- ▷ **Note:** Equivalent to breadth-first search if all step costs are equal (DFS: see below)

Arad



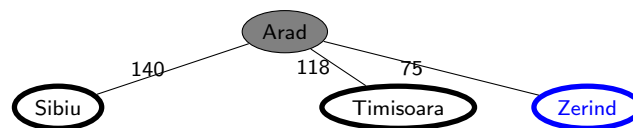
©:Michael Kohlhase

499



Uniform Cost Search: Romania

- ▷ **Idea:** Expand least-cost unexpanded node
- ▷ **Implementation:** fringe is queue ordered by increasing path cost.
- ▷ **Note:** Equivalent to breadth-first search if all step costs are equal (DFS: see below)



©:Michael Kohlhase

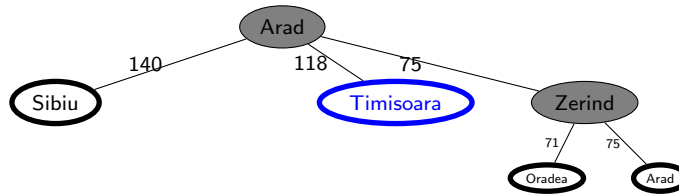
500



Uniform Cost Search: Romania

- ▷ **Idea:** Expand least-cost unexpanded node

- ▷ **Implementation:** fringe is queue ordered by increasing path cost.
- ▷ **Note:** Equivalent to breadth-first search if all step costs are equal (DFS: see below)



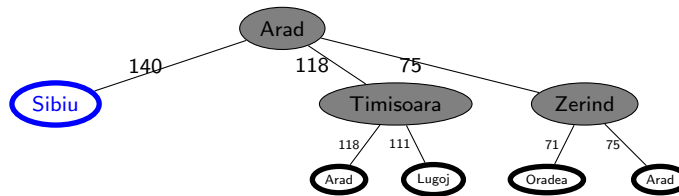
©: Michael Kohlhase

501



Uniform Cost Search: Romania

- ▷ **Idea:** Expand least-cost unexpanded node
- ▷ **Implementation:** fringe is queue ordered by increasing path cost.
- ▷ **Note:** Equivalent to breadth-first search if all step costs are equal (DFS: see below)



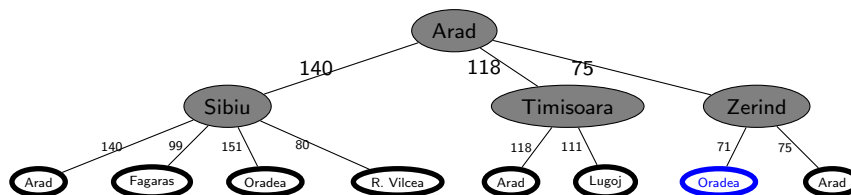
©: Michael Kohlhase

502



Uniform Cost Search: Romania

- ▷ **Idea:** Expand least-cost unexpanded node
- ▷ **Implementation:** fringe is queue ordered by increasing path cost.
- ▷ **Note:** Equivalent to breadth-first search if all step costs are equal (DFS: see below)



©: Michael Kohlhase

503



Note that we must sum the distances to each leaf. That is, we go back to the first level after step

3.

Uniform-cost search: Properties

Complete	Yes (if step costs $\geq \epsilon > 0$)
Time	number of nodes with past-cost less than that of optimal solution
Space	number of nodes with past-cost less than that of optimal solution
Optimal	Yes

©:Michael Kohlhase 504

If step cost is negative, the same situation as in breadth-first search can occur: later solutions may be cheaper than the current one.

If step cost is 0, one can run into infinite branches. UC search then degenerates into depth-first search, the next kind of search algorithm. Even if we have infinite branches, where the sum of step costs converges, we can get into trouble⁸

EdN:8

Worst case is often worse than BF search, because large trees with small steps tend to be searched first. If step costs are uniform, it degenerates to BF search.

Depth-first search

- ▷ **Idea:** Expand deepest unexpanded node
- ▷ **Definition 16.3.2 (Implementation)** **depth first search** is tree search where the *fringe* is organized as a LIFO queue (a stack), i.e. successors go in at front of queue
- ▷ **Note:** Depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

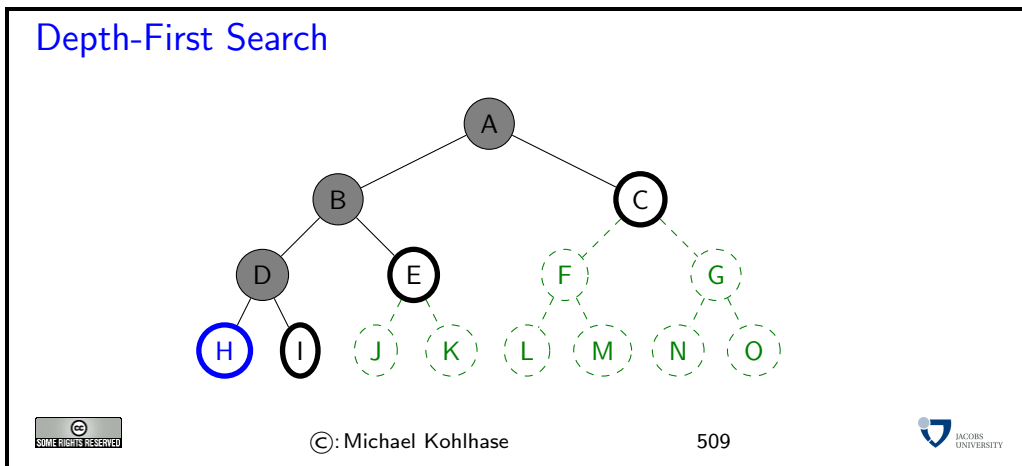
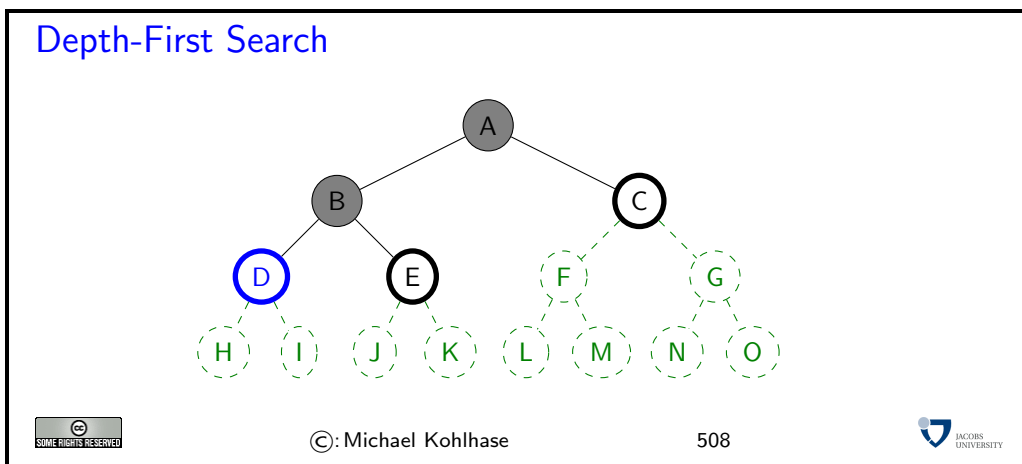
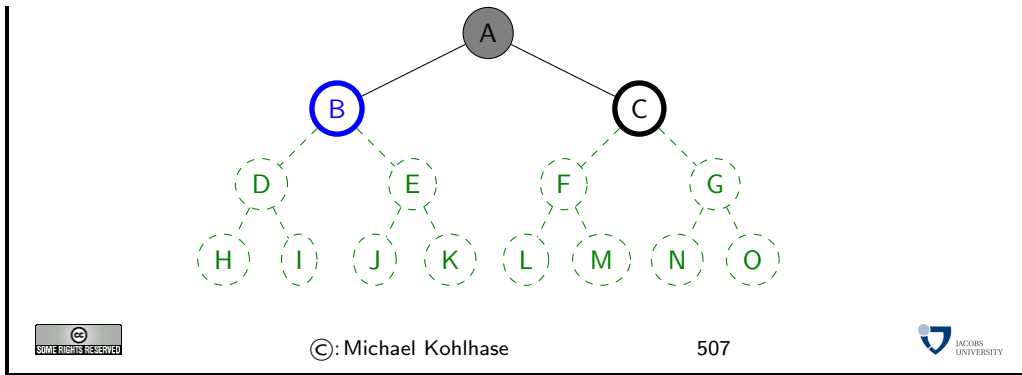
©:Michael Kohlhase 505

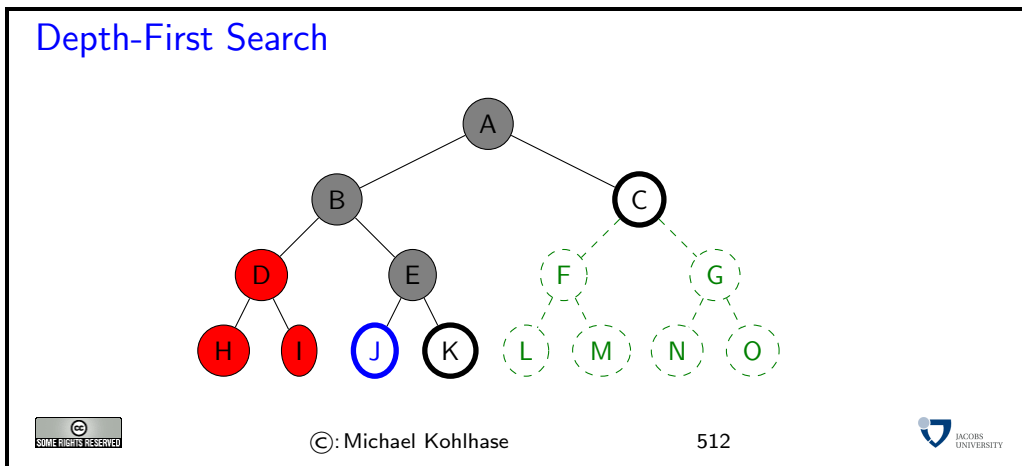
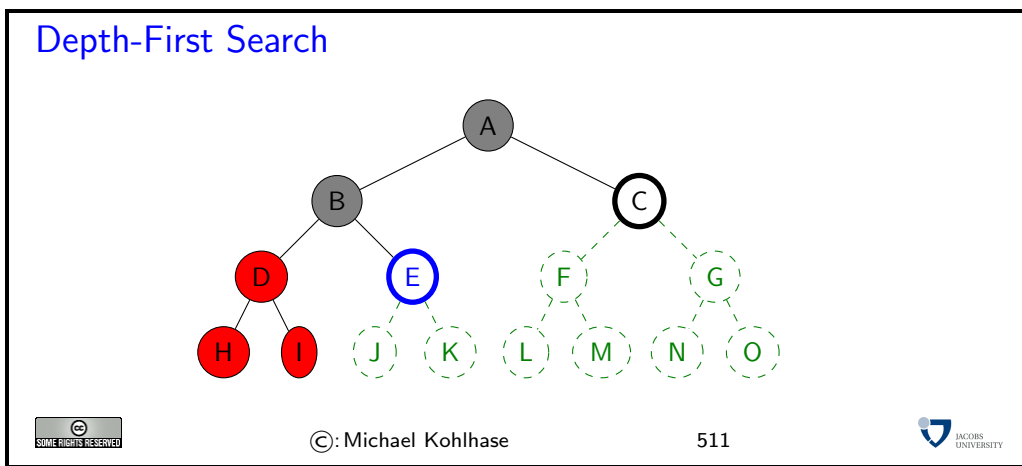
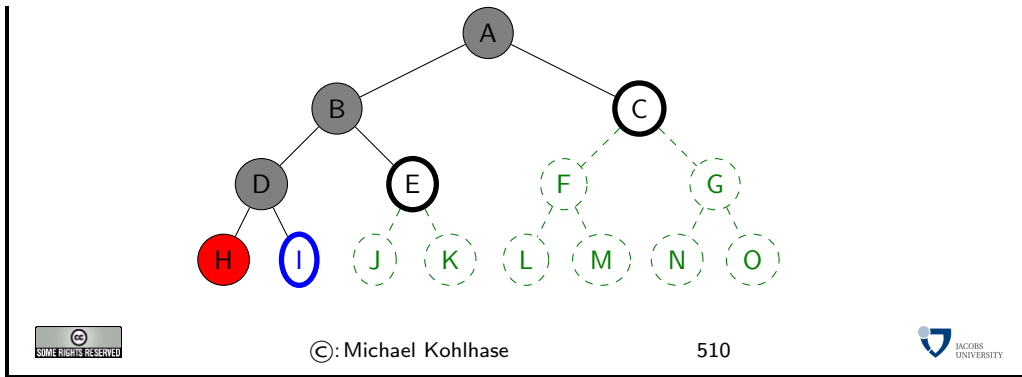
Depth-First Search

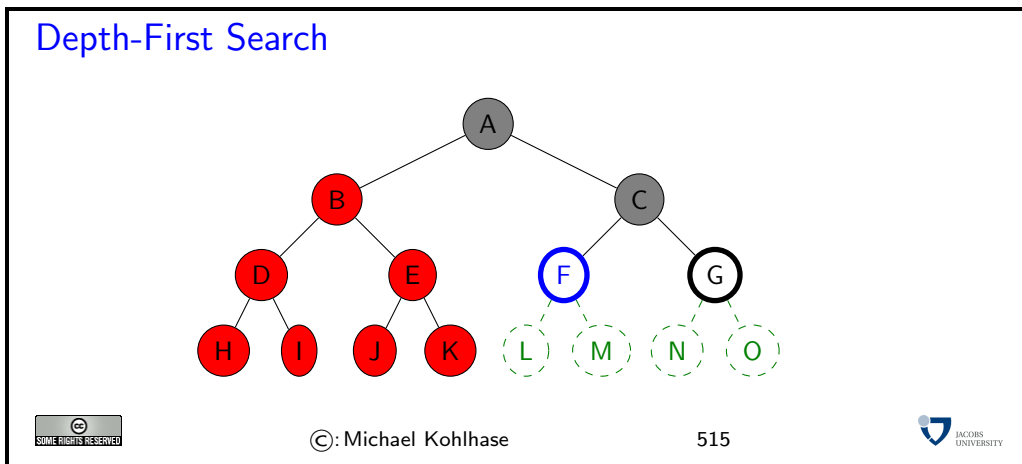
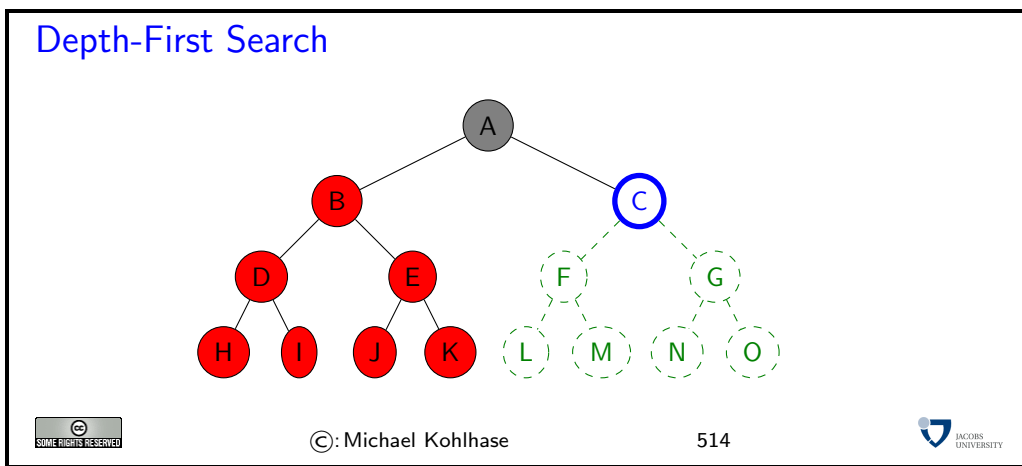
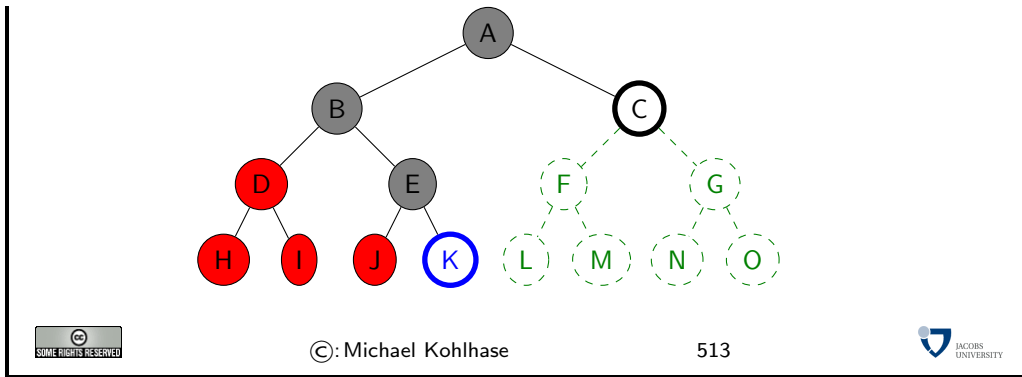
©:Michael Kohlhase 506

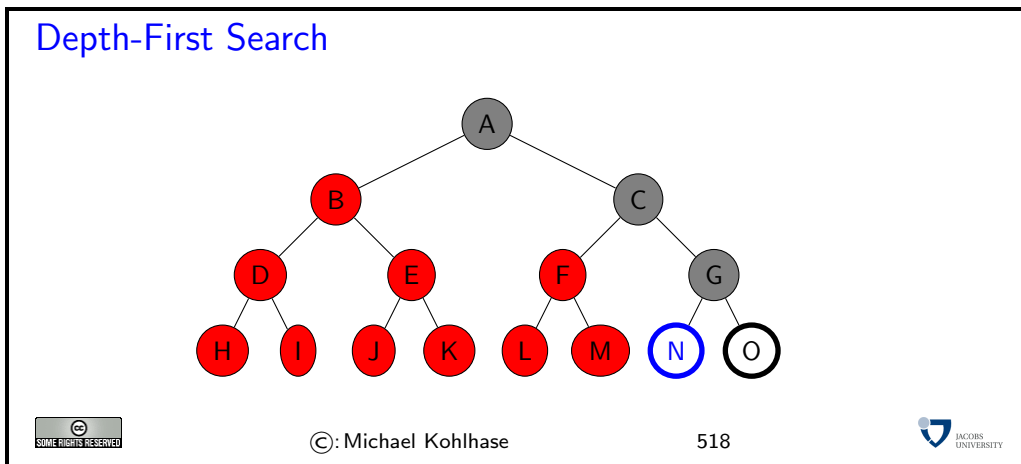
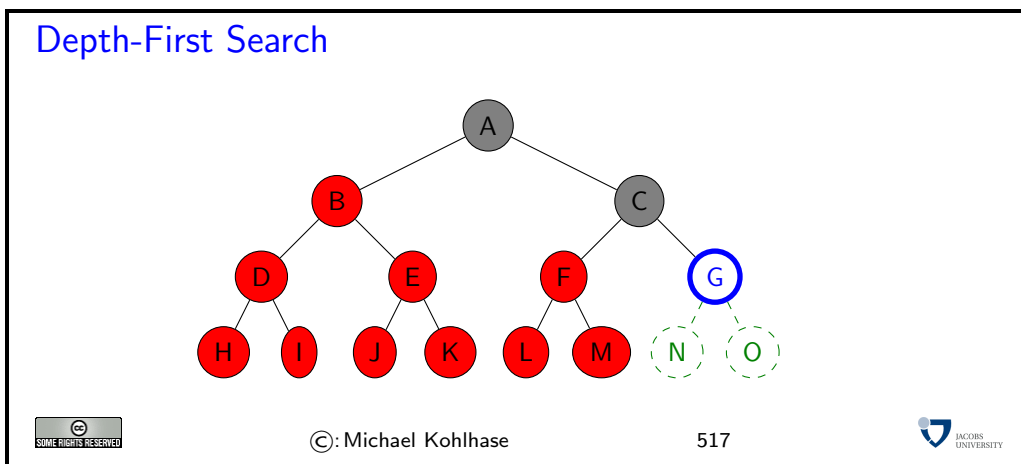
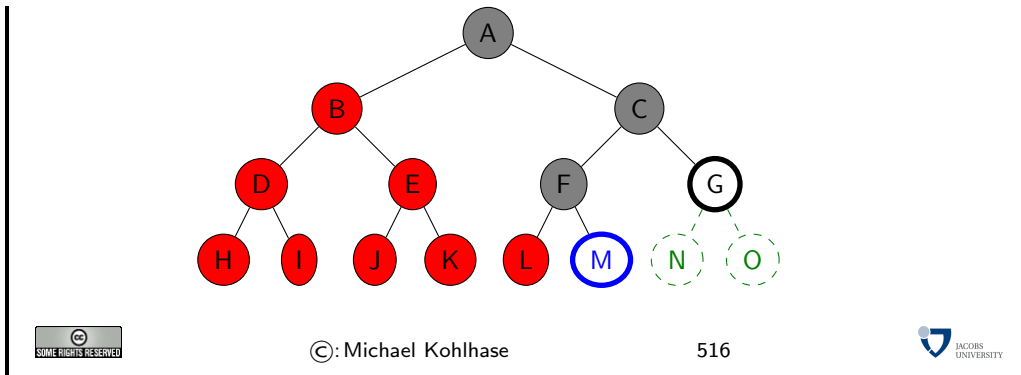
Depth-First Search

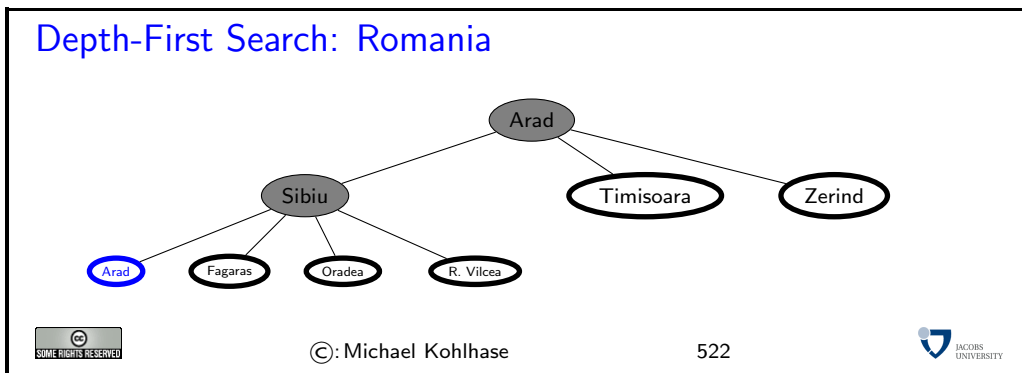
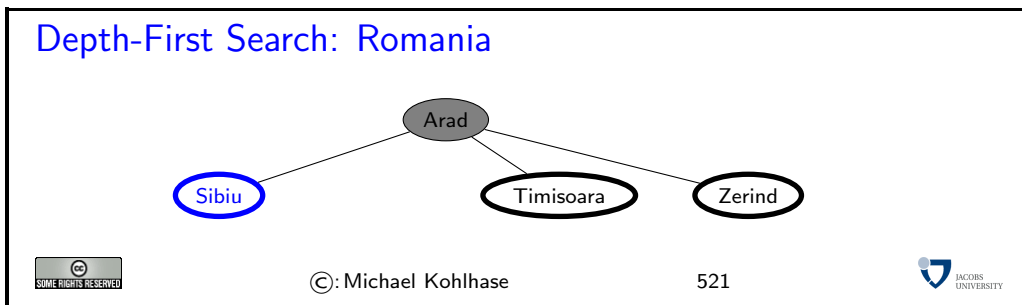
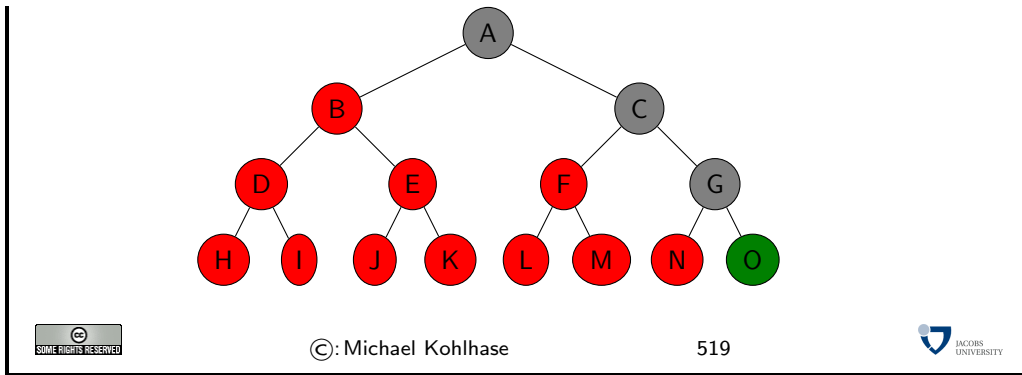
⁸EDNOTE: say how

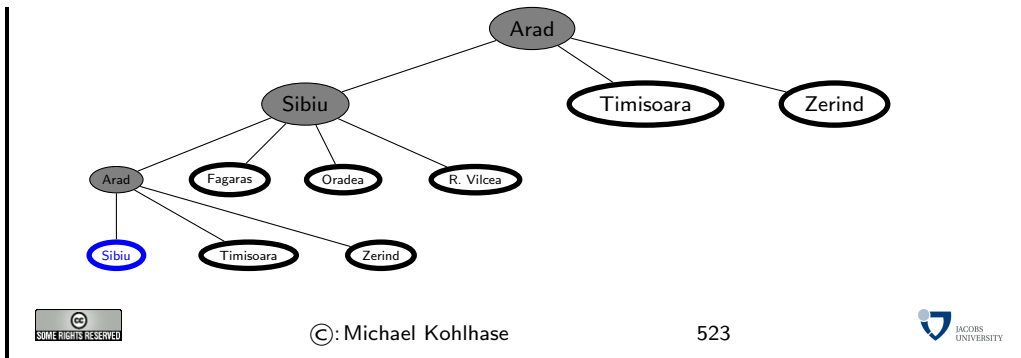












Depth-first search: Properties

Complete	Yes: if state space finite No: if state contains infinite paths or loops
Time	$O(b^m)$ (we need to explore until max depth m in any case!)
Space	$O(b \cdot m)$ (i.e. linear space) (need at most store m levels and at each level at most b nodes)
Optimal	No (there can be many better solutions in the unexplored part of the search tree)

- ▷ **Disadvantage:** Time terrible if m much larger than d .
- ▷ **Advantage:** Time may be much less than breadth-first search if solutions are dense.



Iterative deepening search

- ▷ **Depth-limited search:** Depth-first search with depth limit
- ▷ **Iterative deepening search:** Depth-limit search with ever increasing limits

```

procedure Tree_Search (problem)
  <initialize the search tree using the initial state of problem>
  for depth = 0 to  $\infty$ 
    result := Depth_Limited_search(problem,depth)
    if depth  $\neq$  cutoff return result end if
  end for
end procedure

```



Iterative Deepening Search at Limit Depth 0



Iterative Deepening Search at Limit Depth 1

SOME RIGHTS RESERVED ©: Michael Kohlhase 527 JACOBS UNIVERSITY

Iterative Deepening Search at Limit Depth 2

SOME RIGHTS RESERVED ©: Michael Kohlhase 528 JACOBS UNIVERSITY

Iterative Deepening Search at Limit Depth 3

SOME RIGHTS RESERVED ©: Michael Kohlhase 529 JACOBS UNIVERSITY

Iterative deepening search: Properties

Complete	Yes
Time	$(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d \in O(b^{d+1})$
Space	$O(bd)$
Optimal	Yes (if step cost = 1)

▷ (Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.

▷ Comparison:

Criterion	Breadth-first	Uniform-cost	Depth-first	Iterative deepening
Complete?	Yes*	Yes*	No	Yes
Time	b^{d+1}	$\approx b^d$	b^m	b^d
Space	b^{d+1}	$\approx b^d$	bm	bd
Optimal?	Yes*	Yes	No	Yes



©: Michael Kohlhase

530



Note: To find a solution (at depth d) we have to search the whole tree up to d . Of course since we do not save the search state, we have to re-compute the upper part of the tree for the next level. This seems like a great waste of resources at first, however, iterative deepening search tries to be complete without the space penalties.

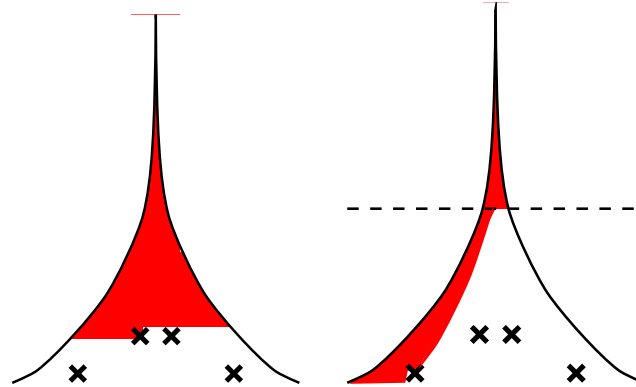
However, the space complexity is as good as depth-first search, since we are using depth-first search along the way. Like in breadth-first search, the whole tree on level d (of optimal solution) is explored, so optimality is inherited from there. Like breadth-first search, one can modify this to incorporate uniform cost search.

As a consequence, variants of iterative deepening search are the method of choice if we do not have additional information.

Comparison

Breadth-first search

Iterative deepening search



©: Michael Kohlhase

531



16.4 Informed Search Strategies

Summary: Uninformed Search/Informed Search



- ▷ Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- ▷ Variety of uninformed search strategies

▷ Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

▷ **Next Step:** Introduce additional knowledge about the problem (informed search)

▷ Best-first-, A^* -search (guide the search by heuristics)

▷ Iterative improvement algorithms

 ©:Michael Kohlhase 532 



16.4.1 Greedy Search

Best-first search

▷ **Idea:** Use an **evaluation function** for each node (estimate of “desirability”) Expand most desirable unexpanded node

▷ **Implementation:** *fringe* is a queue sorted in decreasing order of desirability

▷ **Special cases:** Greedy search, A^* -search

 ©:Michael Kohlhase 533 

This is like UCS, but with evaluation function related to problem at hand replacing the path cost function.

If the heuristics is arbitrary, we expect incompleteness!

Depends on how we measure “desirability”.

Concrete examples follow.

Greedy search



▷ **Definition 16.4.1** A **heuristic** is an evaluation function h on nodes that estimates of cost from n to the nearest goal state.

Idea: Greedy search expands the node that **appears** to be closest to goal

▷ **Example 16.4.2** straight-line distance from to Bucharest.

▷ **Note:** Unlike uniform-cost search the node evaluation function has nothing to do with the nodes explored so far

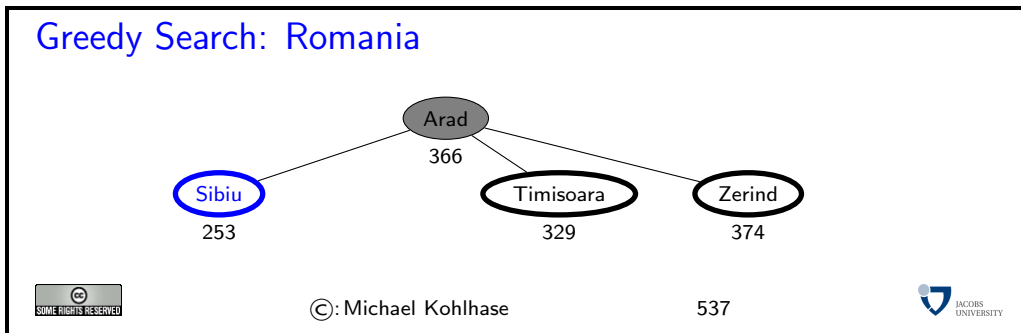
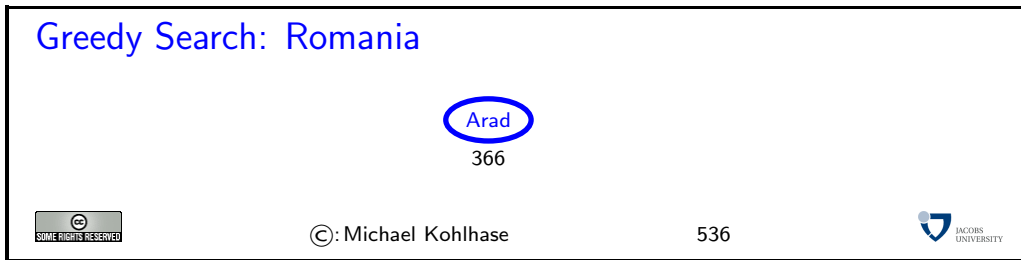
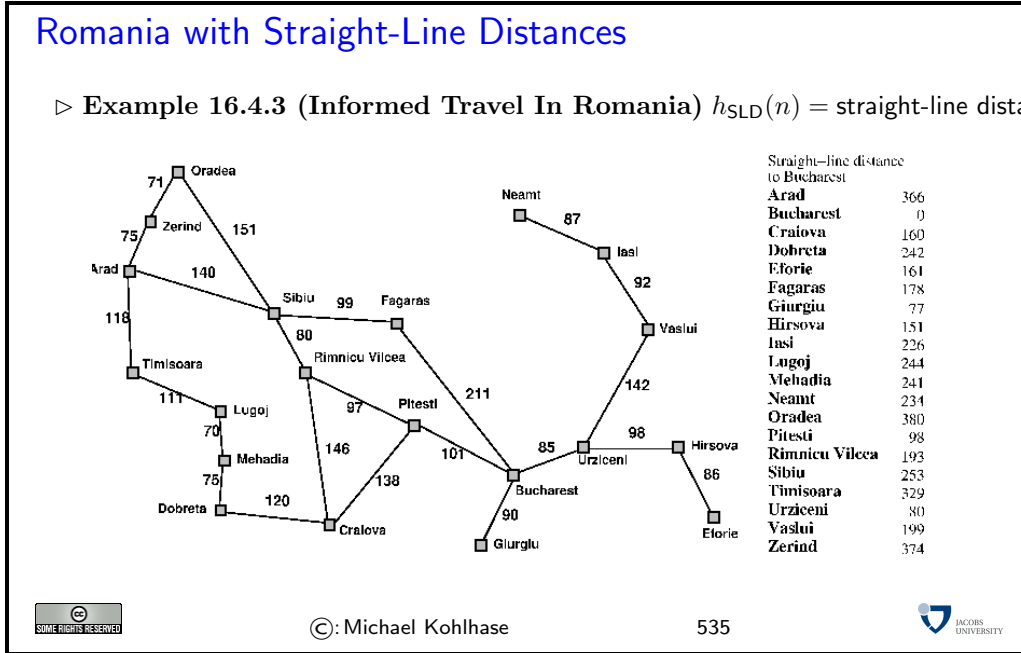
internal search control → external search control
partial solution cost → goal cost estimation

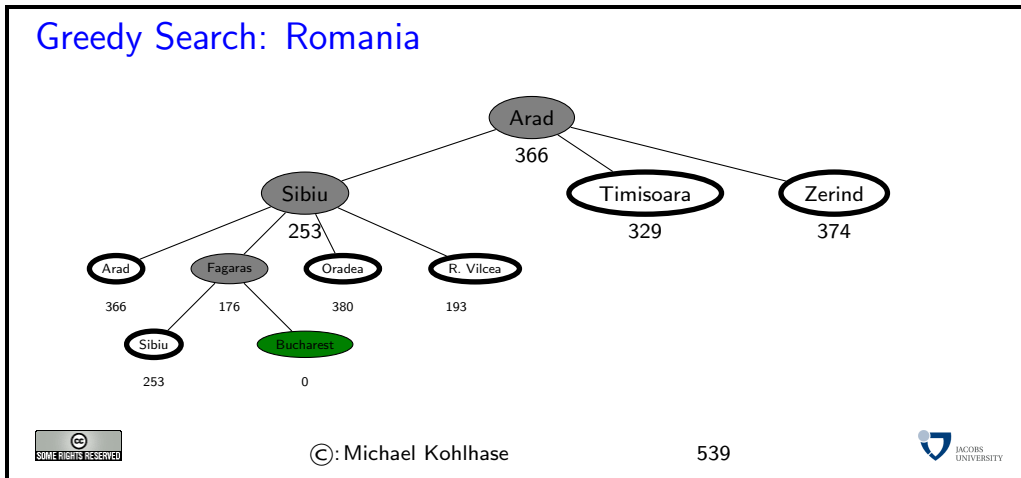
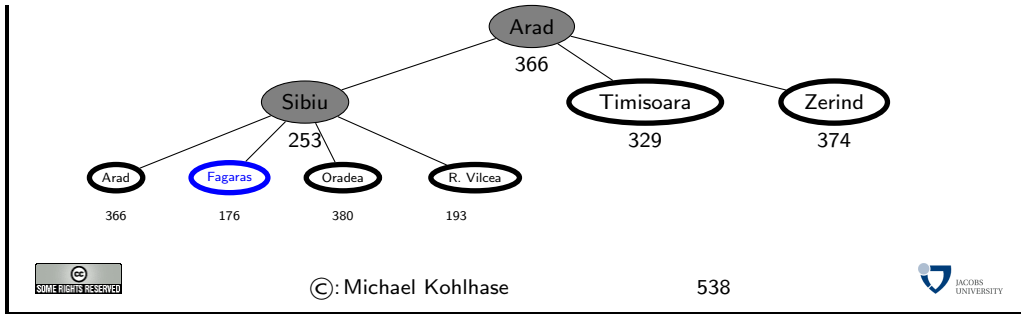
 ©:Michael Kohlhase 534 

In greedy search we replace the *objective* cost to *construct* the current solution with a heuristic or *subjective* measure from which we think it gives a good idea how far we are from a *solution*. Two things have shifted:

- we went from internal (determined only by features inherent in the search space) to an external/heuristic cost

- instead of measuring the cost to build the current partial solution, we estimate how far we are from the desired goal





Greedy search: Properties

- Complete** No: Can get stuck in loops
Complete in finite space with repeated-state checking
- Time** $O(b^m)$
- Space** $O(b^m)$
- Optimal** No

- ▷ **Example 16.4.4** Greedy search can get stuck going from Iasi to Oradea:
Iasi → Neamt → Iasi → Neamt → ...
- ▷ Worst-case time same as depth-first search,
- ▷ Worst-case space same as breadth-first
- ▷ But a good heuristic can give dramatic improvement

©: Michael Kohlhase 540

Greedy Search is similar to UCS. Unlike the latter, the node evaluation function has nothing to do with the nodes explored so far. This can prevent nodes from being enumerated systematically as they are in UCS and BFS.

For completeness, we need repeated state checking as the example shows. This enforces complete enumeration of state space (provided that it is finite), and thus gives us completeness.

Note that nothing prevents from *all* nodes being searched in worst case; e.g. if the heuristic function gives us the same (low) estimate on all nodes except where the heuristic mis-

estimates the distance to be high. So in the worst case, greedy search is even worse than BFS, where d (depth of first solution) replaces m .

The search procedure cannot be optional, since actual cost of solution is not considered.

For both, completeness and optimality, therefore, it is necessary to take the actual cost of partial solutions, i.e. the path cost, into account. This way, paths that are known to be expensive are avoided.

16.4.2 A-Star Search

A^* search

- ▷ **Idea:** Avoid expanding paths that are already expensive (make use of actual cost)

The simplest way to combine heuristic and path cost is to simply add them.

- ▷ **Definition 16.4.5** The **evaluation function** for A^* -search is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost for n and $h(n)$ is the estimated cost to goal from n .
- ▷ Thus $f(n)$ is the estimated total cost of path through n to goal
- ▷ **Definition 16.4.6** Best-First-Search with evaluation function $g + h$ is called **A^* search**.



This works, provided that h does not overestimate the true cost to achieve the goal. In other words, h must be *optimistic* wrt. the real cost h^* . If we are too pessimistic, then non-optimal solutions have a chance.

A^* Search: Admissibility

- ▷ **Definition 16.4.7 (Admissibility of heuristic)** $h(n)$ is called **admissible** if $0 \leq h(n) \leq h^*(n)$ for all nodes n , where $h^*(n)$ is the **true** cost from n to goal. (In particular: $h(G) = 0$ for goal G)
- ▷ **Example 16.4.8** Straight-line distance never overestimates the actual road distance (triangle inequality)
Thus $h_{SLD}(n)$ is admissible.



A^* Search: Admissibility

- ▷ **Theorem 16.4.9** A^* search with admissible heuristic is optimal
 - ▷ **Proof:** We show that sub-optimal nodes are never selected by A^*
- P.1** Suppose a suboptimal goal G has been generated then we are in the following situation:

P.2 Let n be an unexpanded node on a path to an optimal goal O , then

$f(G) = g(G)$	since $h(G) = 0$
$g(G) > g(O)$	since G suboptimal
$g(O) = g(n) + h^*(n)$	n on optimal path
$g(n) + h^*(n) \geq g(n) + h(n)$	since h is admissible
$g(n) + h(n) = f(n)$	

P.3 Thus, $f(G) > f(n)$ and *astarSearch* never selects G for expansion. \square

SOME RIGHTS RESERVED ©: Michael Kohlhase 543

A* Search Example

Arad
366=0+366

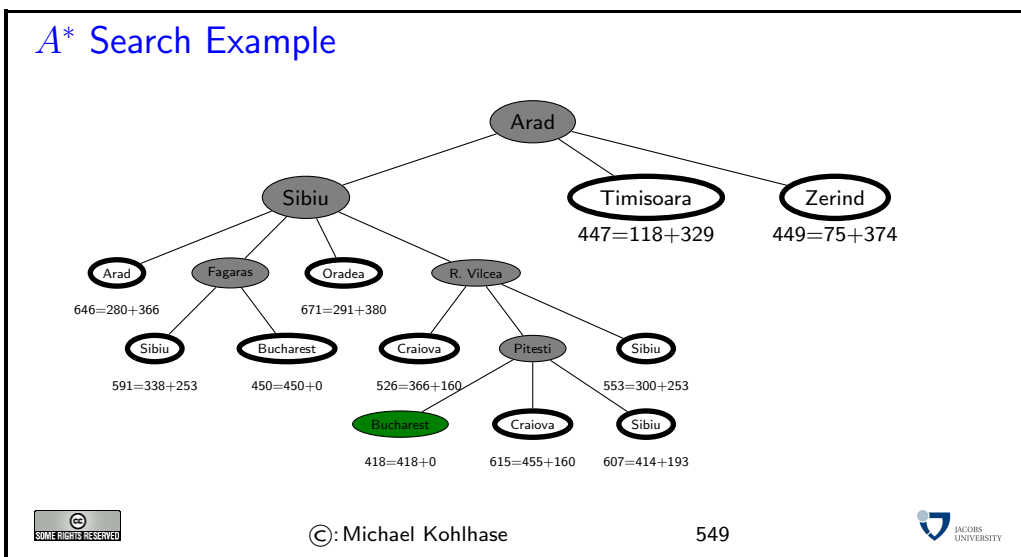
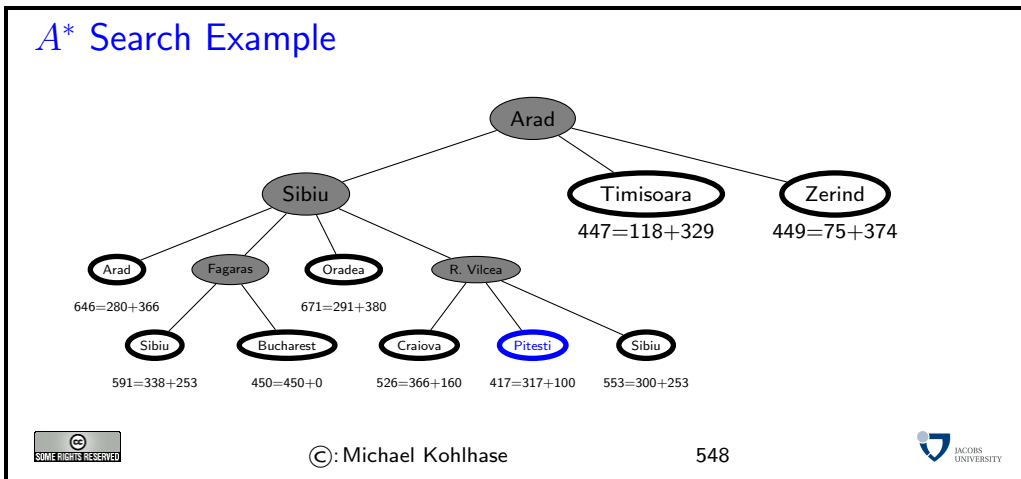
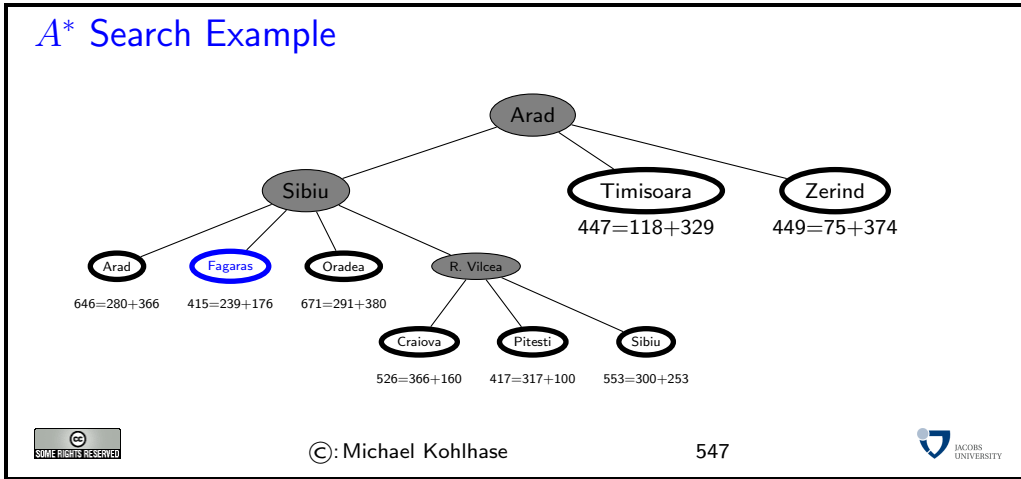
SOME RIGHTS RESERVED ©: Michael Kohlhase 544

A* Search Example

SOME RIGHTS RESERVED ©: Michael Kohlhase 545

A* Search Example

SOME RIGHTS RESERVED ©: Michael Kohlhase 546



A* search: *f*-contours

▷ A^* gradually adds “ f -contours” of nodes

©: Michael Kohlhase 550

A^* search: Properties

Complete	Yes (unless there are infinitely many nodes n with $f(n) \leq f(0)$)
Time	Exponential in [relative error in $h \times$ length of solution]
Space	Same as time (variant of BFS)
Optimal	Yes

▷ A^* expands all (some/no) nodes with $f(n) < h^*(n)$

▷ The run-time depends on how good we approximated the real cost h^* with h .

©: Michael Kohlhase 551

16.4.3 Finding Good Heuristics

Since the availability of admissible heuristics is so important for informed search (particularly for A^*), let us see how such heuristics can be obtained in practice. We will look at an example, and then derive a general procedure from that.

Admissible heuristics: Example 8-puzzle

7	2	4
5		6
8	3	1

1	2	3
4	5	6
7	8	

Start State Goal State

▷ **Example 16.4.10** Let $h_1(n)$ be the number of misplaced tiles in node n
 $(h_1(S) = 6)$

▷ **Example 16.4.11** Let $h_2(n)$ be the total manhattan distance from desired location of each tile. $(h_2(S) = 2 + 0 + 3 + 1 + 0 + 1 + 3 + 4 = 14)$

▷ **Observation 16.4.12 (Typical search costs)** $(IDS \hat{=} \textit{iterative deepening search})$

nodes explored	IDS	$A^*(h_1)$	$A^*(h_2)$
$d = 14$	3,473,941	539	113
$d = 24$	too many	39,135	1,641



©: Michael Kohlhase

552



Dominance

▷ **Definition 16.4.13** Let h_1 and h_2 be two admissible heuristics we say that h_2 **dominates** h_1 if $h_2(n) \geq h_1(n)$ or all n .

▷ **Theorem 16.4.14** If h_2 dominates h_1 , then h_2 is better for search than h_1 .



©: Michael Kohlhase

553



Relaxed problems

▷ **Finding good admissible heuristics is an art!**

▷ **Idea:** Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem.

▷ **Example 16.4.15** If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then we get heuristic h_1 .

▷ **Example 16.4.16** If the rules are relaxed so that a tile can move to *any adjacent square*, then we get heuristic h_2 .

▷ **Key point:** The optimal solution cost of a relaxed problem is not greater than the optimal solution cost of the real problem



©: Michael Kohlhase

554



Relaxation means to remove some of the constraints or requirements of the original problem, so that a solution becomes easy to find. Then the cost of this easy solution can be used as an optimistic approximation of the problem.

16.5 Local Search

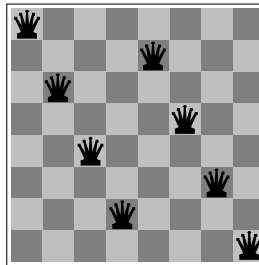
Local Search Problems

▷ **Idea:** Sometimes the path to the solution is irrelevant

▷ **Example 16.5.1 (8 Queens Problem)**

Place 8 queens on a chess board, so that no two queens threaten each other.

▷ This problem has various solutions, e.g. the one on the right



▷ **Definition 16.5.2** A **local search** algorithm is a search algorithm that operates on a single state, the **current state** (rather than multiple paths). (advantage: **constant space**)

▷ Typically local search algorithms only move to successors of the current state, and do not retain search paths.

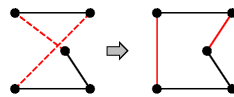
▷ Applications include: integrated circuit design, factory-floor layout, job-shop scheduling, portfolio management, fleet deployment,...



Local Search: Iterative improvement algorithms

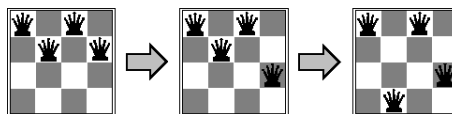
▷ **Definition 16.5.3 (Traveling Salesman Problem)** Find shortest trip through set of cities such that each city is visited exactly once.

▷ **Idea:** Start with any complete tour, perform pairwise exchanges



▷ **Definition 16.5.4 (n -queens problem)** Put n queens on $n \times n$ board such that no two queens in the same row, columns, or diagonal.

▷ **Idea:** Move a queen to reduce number of conflicts



Hill-climbing (gradient ascent/descent)

▷ **Idea:** Start anywhere and go in the direction of the steepest ascent.

▷ Depth-first search with heuristic and w/o memory

```
procedure Hill-Climbing (problem) (* a state that is a local minimum *)
  local current, neighbor (* nodes *)
```

```

current := Make-Node(Initial-State[problem])
loop
  neighbor := <a highest-valued successor of current>
  if Value[neighbor] < Value[current]
    return [current]
    current := neighbor
  end if
end loop
end procedure

```

- ▷ Like starting anywhere in search tree and making a heuristically guided DFS.
- ▷ Works, if solutions are dense and local maxima can be escaped.



©: Michael Kohlhase

557



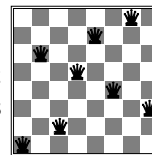
In order to understand the procedure on a more intuitive level, let us consider the following scenario: We are in a dark landscape (or we are blind), and we want to find the highest hill. The search procedure above tells us to start our search anywhere, and for every step first feel around, and then take a step into the direction with the steepest ascent. If we reach a place, where the next step would take us down, we are finished.

Of course, this will only get us into local maxima, and has no guarantee of getting us into global ones (remember, we are blind). The solution to this problem is to re-start the search at random (we do not have any information) places, and hope that one of the random jumps will get us to a slope that leads to a global maximum.

Example Hill-Climbing with 8 Queens

- ▷ **Idea:** Heuristic function h is number of queens that threaten each other.
- ▷ **Example 16.5.5** An 8-queens state with heuristic cost estimate $h = 17$ showing h -values for moving a queen within its column
- ▷ **Problem:** The state space has local minima. e.g. the board on the right has $h = 1$ but every successor has $h > 1$.

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	13	16	13	16	
14	17	15	14	14	16	16	
17	16	18	15	15	15	15	
18	14	15	15	14	14	16	
14	14	13	17	12	14	12	18



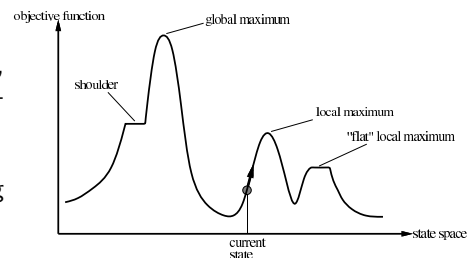
©: Michael Kohlhase

558



Hill-climbing

- ▷ **Problem:** Depending on initial state, can get stuck on local maxima/minima and plateaux
- ▷ "Hill-climbing search is like climbing Everest in thick fog with amnesia"



- ▷ **Idea:** Escape local maxima by allowing some "bad" or random moves.

- ▷ **Example 16.5.6** local search, simulated annealing...
- ▷ **Properties:** All are incomplete, non-optimal.
- ▷ Sometimes performs well in practice (if (optimal) solutions are dense)



©: Michael Kohlhase

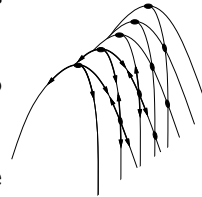
559



Recent work on hill-climbing algorithms tries to combine complete search with randomization to escape certain odd phenomena occurring in statistical distribution of solutions.

Simulated annealing (Idea)

- ▷ **Definition 16.5.7** Ridge s are ascending successions of local maxima
- ▷ **Problem:** They are extremely difficult to navigate for local search algorithms
- ▷ **Idea:** Escape local maxima by allowing some “bad” moves, but gradually decrease their size and frequency
- ▷ Annealing is the process of heating steel and let it cool gradually to give it time to grow an optimal cristal structure.
- ▷ Simulated Annealing is like shaking a ping-pong ball occasionally on a bumpy surface to free it. (so it does not get stuck)
- ▷ Devised by Metropolis et al., 1953, for physical process modelling
- ▷ Widely used in VLSI layout, airline scheduling, etc.



©: Michael Kohlhase

560



Simulated annealing (Implementation)

```

procedure Simulated-Annealing (problem,schedule) (* a solution state *)
  local node, next (* nodes*)
  local T (*a ‘temperature’ controlling prob. of downward steps *)
  current := Make-Node(Initial-State[problem])
  for t :=1 to ∞
    T := schedule[t]
    if T = 0 return current end if
    next := <a randomly selected successor of current>
    Δ(E) := Value[next]-Value[current]
    if Δ(E) > 0 current := next
    else
      current := next <only with probability> eΔ(E)/T
    end if
  end for
end procedure

```

a problem schedule is a mapping from time to “temperature”



©: Michael Kohlhase

Properties of simulated annealing

- ▷ At fixed “temperature” T , state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{-\frac{E(x)}{kT}}$$

T decreased slowly enough \implies always reach best state x^* because $\frac{e^{-\frac{E(x^*)}{kT}}}{e^{-\frac{E(x)}{kT}} = e^{-\frac{E(x^*) - E(x)}{kT}}} \gg 1$ for small T .

- ▷ Is this necessarily an interesting guarantee?



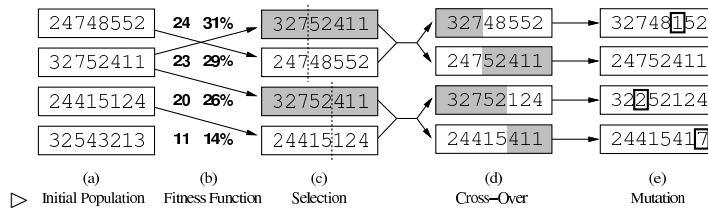
Local beam search

- ▷ **Idea:** Keep k states instead of 1; choose top k of all their successors
- ▷ Not the same as k searches run in parallel! (Searches that find good states recruit other searches to join them)
- ▷ **Problem:** quite often, all k states end up on same local hill
- ▷ **Idea:** Choose k successors randomly, biased towards good ones. (Observe the close analogy to natural selection!)



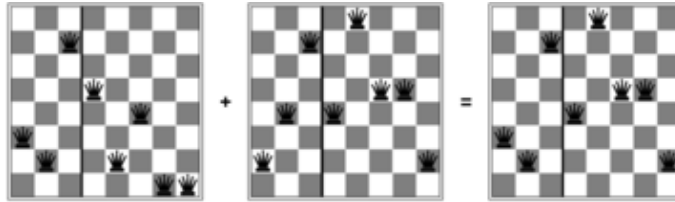
Genetic algorithms (very briefly)

- ▷ **Idea:** Use local beam search (keep a population of k) randomly modify population (mutation) generate successors from pairs of states (sexual reproduction) optimize a fitness function (survival of the fittest)



Genetic algorithms (continued)

- ▷ **Problem:** Genetic Algorithms require states encoded as strings (GPs use programs)
- ▷ Crossover helps iff substrings are meaningful components

▷ **Example 16.5.8 (Evolving 8 Queens)**

▷ GAs \neq evolution: e.g., real genes encode replication machinery!



Chapter 17

Logic Programming

We will now learn a new programming paradigm: “logic programming” (also called “Declarative Programming”), which is an application of the search techniques we looked at last, and the logic techniques. We are going to study ProLog (the oldest and most widely used) as a concrete example of the ideas behind logic programming.

17.1 Introduction to Logic Programming and PROLOG

Logic Programming is a programming style that differs from functional and imperative programming in the basic procedural intuition. Instead of transforming the state of the memory by issuing instructions (as in imperative programming), or computing the value of a function on some arguments, logic programming interprets the program as a body of knowledge about the respective situation, which can be queried for consequences. This is actually a very natural intuition; after all we only run (imperative or functional) programs if we want some question answered.

Logic Programming

- ▷ **Idea:** Use logic as a programming language!
- ▷ We state what we know about a problem (the program) and then ask for results (what the program would compute)

▷ Example 17.1.1

Program	Leibniz is human Sokrates is is human Sokrates is a greek Every human is fallible	$x + 0 = x$ If $x + y = z$ then $x + s(y) = s(z)$ 3 is prime
Query	Are there fallible greeks?	is there a z with $s(s(0)) + s(0) = z$
Answer	Yes, Sokrates!	yes $s(s(0))$

- ▷ **How to achieve this?:** Restrict the logic calculus sufficiently that it can be used as computational procedure.
- ▷ **Slogan:** Computation = Logic + Control ([Kowalski '73])
- ▷ We will use the programming language ProLog as an example



Of course, this the whole point of writing down a knowledge base (a program with knowledge

about the situation), if we do not have to write down *all* the knowledge, but a (small) subset, from which the rest follows. We have already seen how this can be done: with logic. For logic programming we will use a logic called “first-order logic” which we will not formally introduce here. We have already seen that we can formulate propositional logic using terms from an abstract data type instead of propositional variables – recall Definition 183. For our purposes, we will just use terms with variables instead of the ground terms used there.

Representing a Knowledge base in ProLog

- ▷ **Definition 17.1.2** Fix an abstract data type $\langle \{\mathbb{B}, \dots\}, \{\dots\} \rangle$, then we call all constructor terms of sort \mathbb{B} **ProLog terms**.
- ▷ A knowledge base is represented (symbolically) by a set of facts and rules.
- ▷ **Definition 17.1.3** A **fact** is a statement written as a term that is unconditionally true of the domain of interest. (write with a term followed by a “.”)
- ▷ **Example 17.1.4** We can state that Mia is a woman as `woman(mia)`.
- ▷ **Definition 17.1.5** A **rule** states information that is *conditionally* true in the domain.
- ▷ **Definition 17.1.6** Facts and rules are collectively called **clauses**, a **ProLog program** consists of a list of clauses.
- ▷ **Example 17.1.7** Write “something is a car if it has a motor and four wheels” as
`car(X) :- has_motor(X),has_wheels(X,4)`. (variables are upper-case)
 this is just an ASCII notation for $m(x) \wedge w(x,4) \Rightarrow car(x)$
- ▷ **Definition 17.1.8** The **knowledge base** given by ProLog program is that set of facts that can be derived from it by Modus Ponens (MP), $\wedge I$ and instantiation.

$$\frac{A \quad A \Rightarrow B}{B} \text{MP} \qquad \frac{A \quad B}{A \wedge B} \wedge I \qquad \frac{A}{[B/X](A)} \text{Subst}$$




To gain an intuition for this quite abstract definition let us consider a concrete knowledge base about cars. Instead of writing down everything we know about cars, we only write down that cars are motor vehicles with four wheels and that a particular object c has a motor and four wheels. We can see that the fact that c is a car can be derived from this. Given our definition of a knowledge base as the deductive closure of the facts and rules explicitly written down, the assertion that c is a car is in the “induced knowledge base”, which is what we are after.

Knowledge Base (Example)


- ▷ **Example 17.1.9** `car(c)` is in the knowledge base generated by


```
has_motor(c).
has_wheels(c,4).
car(X) :- has_motor(X),has_wheels(X,4).
```

$$\frac{\frac{m(c) \quad w(c,4)}{m(c) \wedge w(c,4)} \wedge I \quad \frac{m(x) \wedge w(x,4) \Rightarrow car(x)}{m(c) \wedge w(c,4) \Rightarrow car(c)} \text{Subst}}{car(c)} \text{MP}$$


©: Michael Kohlhase

568



In this very simple example $car(c)$ is about the only fact we can derive, but in general, knowledge bases can be infinite (we will see examples below)

As knowledge bases can be infinite, we cannot pre-compute them. Instead, logic programming languages compute fragments of the knowledge base by need; i.e. whenever a user wants to check membership; we call this approach querying: the user enters a query term and the system answers **yes** or **no**. This answer is computed in a depth-first search process.


Querying the Knowledge base

- ▷ **Idea:** We want to see whether a term is in the knowledge base.
- ▷ **Definition 17.1.10** A **query** is a list of ProLog terms called **goals**. Write as $?- A_1, \dots, A_n$, if A_i are terms.
- ▷ **Problem:** Knowledge bases can be big and even infinite.
- ▷ **Example 17.1.11** The the knowledge base induced by the program


```
nat(zero).
nat(s(X)) :- nat(X).
```


 contains the facts $\text{nat}(\text{zero}), \text{nat}(\text{s}(\text{zero})), \text{nat}(\text{s}(\text{s}(\text{zero}))), \dots$
- ▷ **Idea:** interpret this as a search problem.
 - ▷ state = tuple of goals; goal state = empty list (of goals).
 - ▷ $\text{next}(\langle G, R_1, \dots, R_l \rangle) := \langle \sigma(B_1), \dots, \sigma(B_m, R_1, \dots, R_l) \rangle$ (**backchaining**) if there is a rule $H :- B_1, \dots, B_m$. and a substitution σ with $\sigma(H) = \sigma(G)$.
- ▷ **Example 17.1.12 (Continuing Example 17.1.11)** $?- \text{nat}(\text{s}(\text{s}(\text{zero})))$.


```
?- nat(s(s(zero))).
?- nat(s(zero)).
?- nat(zero).
Yes
```



©: Michael Kohlhase

569



This search process has as action the backchaining rule, which is really just a clever combination of the inference rules from Definition 17.1.8 applied backwards.

The backchaining rule takes the current goal G (the first one) and tries to find a substitution σ and a head H in a program clause, such that $\sigma(G) = \sigma(H)$. If such a clause can be found, it replaces G with the body of the rule instantiated by σ .

Note that backchaining replaces the current query with the body of the rule suitably instantiated. For rules with a body long this extends the list of current goals, but for facts (rules without a body), backchaining shortens the list of current goals. Once there are no goals left, the ProLog interpreter finishes and signals success by issuing the string **Yes**.

If no rules match the current goal, then the interpreter terminates and signals failure with the string `No`.

Querying the Knowledge base: Failure

- ▷ If no instance of the statement in a query can be derived from the knowledge base, then the ProLog interpreter reports failure.

```
?- nat(s(s(0))).
?- nat(s(0)).
?- nat(0).
FAIL
No
```



©: Michael Kohlhase

570



We can extend querying from simple `yes/no` answers to programs that return values by simply using variables in queries. In this case, the ProLog interpreter returns a substitution.

Querying the Knowledge base: Answer Substitutions

- ▷ If a query contains variables, then ProLog will return an **answer substitution**.

```
▷ has_wheels(mybmw,4).
  has_motor(mybmw).
  car(X):-has_wheels(X,4),has_motor(X).
  ?- car(Y) % query
  ?- has_wheels(Y,4),has_motor(Y). % substitution X = Y
  ?- has_motor(mybmw). % substitution Y = mybmw
  Y = mybmw % answer substitution
  Yes
```



©: Michael Kohlhase

571



In the Example 571 the first backchaining step binds the variable `X` to the query variable `Y`, which gives us the two subgoals `has_wheels(Y,4)`, `has_motor(Y)`. which again have the query variable `Y`. The next backchaining step binds this to `mybmw`, and the third backchaining step exhausts the subgoals. So the query succeeds with the (overall) substitution `[mybmw/Y]`, which is reported by the ProLog interpreter as `Y = mybmw`.

With this setup, we can already do the “fallible Greeks” example from the introduction.

PROLOG: Are there Fallible Greeks?

- ▷ Program:

```
human(sokrates).
human(leibniz).
greek(sokrates).
fallible(X) :- human(X).
```

- ▷ Example 17.1.13 (Query) `?-fallible(X),greek(X)`.

- ▷ Answer substitution: `[sokrates/X]`



The examples above mostly highlight the use of ProLog as a search-based query mechanism for the knowledge induced by the program. We will now show that see that this mechanism can be used as a programming language. For that we first specify the search procedure used by the ProLog interpreter in detail.

17.2 Programming as Search

In this section, we want to really use ProLog as a programming language, so let use first get our tools set up.

We will now discuss how to use a ProLog interpreter to get to know the language. The SWI ProLog interpreter can be downloaded from <http://www.swi-prolog.org/>. To start the ProLog interpreter with `pl` or `prolog` or `swipl` from the shell. The SWI manual is available at <http://www.swi-prolog.org/pldoc/>

We will introduce working with the interpreter using unary natural numbers as examples: we first add the `fact1` to the knowledge base

```
unat(zero).
```

which asserts that the predicate `unat2` is `true` on the term `zero`. Generally, we can add a fact to the knowledge base either by writing it into a file (e.g. `example.pl`) and then “consulting it” by writing one of the following commands into the interpreter:

```
[example]
consult('example.pl').
```

or by directly typing

```
assert(unat(zero)).
```

into the ProLog interpreter. Next tell ProLog about the following rule

```
assert(unat(suc(X)) :- unat(X)).
```

which gives the ProLog runtime an initial (infinite) knowledge base, which can be queried by

```
?- unat(suc(suc(zero))).
Yes
```

Running ProLog in an `emacs` window is incredibly nicer than at the command line, because you can see the whole history of what you have done. Its better for debugging too. If you’ve never used `emacs` before, it still might be nicer, since its pretty easy to get used to the little bit of `emacs` that you need. (Just type “`emacs \&`” at the UNIX command line to run it; if you are on a remote terminal like `putty`, you can use “`emacs -nw`”).

If you don’t already have a file in your home directory called “`.emacs`” (note the dot at the front), create one and put the following lines in it. Otherwise add the following to your existing `.emacs` file:

```
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
(setq prolog-program-name "swipl") ; or whatever the prolog executable name is
(add-to-list 'auto-mode-alist '("\\.pl$" . prolog-mode))
```

¹for “unary natural numbers”; we cannot use the predicate `nat` and the constructor functions here, since their meaning is predefined in ProLog

²for “unary natural numbers”.

The file `prolog.el`, which provides `prolog-mode` should already be installed on your machine, otherwise download it at <http://turing.ubishops.ca/home/bruda/emacs-prolog/>

Now, once you're in `emacs`, you will need to figure out what your “meta” key is. Usually its the alt key. (Type “control” key together with “h” to get help on using `emacs`). So you'll need a “meta-X” command, then type “run-prolog”. In other words, type the meta key, type “x”, then there will be a little window at the bottom of your `emacs` window with “M-x”, where you type `run-prolog`³. This will start up the SWI ProLog interpreter, ...et voilà!

The best thing is you can have two windows “within” your `emacs` window, one where you're editing your program and one where you're running ProLog. This makes debugging easier.

Depth-First Search with Backtracking

- ▷ So far, all the examples led to direct success or to failure. (simpl. KB)
- ▷ **Definition 17.2.1 (ProLog Search Procedure)** top-down, left-right depth-first (**backtracking**) search, concretely
 - ▷ Work on the subgoals in left-right order.
 - ▷ match first query with the head literals of the clauses in the program in top-down order.
 - ▷ if there are no matches, fail and backtrack to the (chronologically) last **backtrack point**.
 - ▷ otherwise backchain on the first match, keep the other matches in mind for backtracking via **backtrack point** s.
- ▷ We can force backtracking to get more solutions by typing `;`.



©: Michael Kohlhase

573



Note: We have seen Definition 16.3.2 that depth-first search has the problem that it can go into loops. And in fact this is a necessary feature and not a bug for a programming language: we need to be able to write non-terminating programs, since the language would not be Turing-complete otherwise. The argument can be sketched as follows: we have seen that for Turing machines the **halting problem** is undecidable. So if all ProLog programs were terminating, then ProLog would be weaker than Turing machines and thus not **Turing complete**.

We will now fortify our intuition about the ProLog search procedure by an example that extends the setup from Example 571 by a new choice of a vehicle that could be a car (if it had a motor).

Backtracking by Example



```
has_wheels(mytricycle,3).
has_wheels(myrollerblade,3).
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,3),has_motor(X). % cars sometimes have three wheels
car(X):-has_wheels(X,4),has_motor(X). % and sometimes four.
?- car(Y).
?- has_wheels(Y,3),has_motor(Y). % backtrack point 1
Y = mytricycle % backtrack point 2
?- has_motor(mytricycle).
FAIL % fails, backtrack to 2
Y = myrollerblade % backtrack point 2
?- has_motor(myrollerblade).
FAIL % fails, backtrack to 1
?- has_wheels(Y,4),has_motor(Y).
```

³Type “control” key together with “h” then press “m” to get an exhaustive mode help.

```

Y = mybmw
?- has_motor(mybmw).
Y=mybmw
Yes

```

 ©: Michael Kohlhase 574 

In general, a ProLog rule of the form $A:-B,C$ reads as *A, if B and C*. If we want to express *A if B or C*, we have to express this two separate rules $A:-B$ and $A:-C$ and leave the choice which one to use to the search procedure.

In Example 574 we indeed have two clauses for the predicate `car/1`; one each for the cases of cars with three and four wheels. As the three-wheel case comes first in the program, it is explored first in the search process.

Recall that at every point, where the ProLog interpreter has the choice between two clauses for a predicate, chooses the first and leaves a [backtrack point](#). In Example 574 this happens first for the predicate `car/1`, where we explore the case of three-wheeled cars. The ProLog interpreter immediately has to choose again – between the tricycle and the rollerblade, which both have three wheels. Again, it chooses the first and leaves a [backtrack point](#). But as tricycles do not have motors, the subgoal `has_motor(mytricycle)` fails and the interpreter backtracks to the chronologically nearest [backtrack point](#) (the second one) and tries to fulfill `has_motor(myrollerblade)`. This fails again, and the next backtrack point is point 1 – note the stack-like organization of backtrack points which is in keeping with the depth-first search strategy – which chooses the case of four-wheeled cars. This ultimately succeeds as before with `y=mybmw`.

We now turn to a more classical programming task: computing with numbers. Here we turn to our initial example: adding unary natural numbers. If we can do that, then we have to consider ProLog a programming language.

Can We Use This For Programming?

- ▷ **Question:** What about functions? E.g. the addition function?
- ▷ **Question:** We do not have (binary) functions, in ProLog
- ▷ **Idea (back to math):** use a three-place predicate.

Example 17.2.2 `add(X,Y,Z)` stands for $X+Y=Z$

- ▷ Now we can directly write the recursive equations $X + 0 = X$ (base case) and $X + s(Y) = s(X + Y)$ into the knowledge base.

```

add(X,zero,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).



```

- ▷ similarly with multiplication and exponentiation.

```

mult(X,o,o).
mult(X,s(Y),Z) :- mult(X,Y,W), add(X,W,Z).
expt(X,o,s(o)).
expt(X,s(Y),Z) :- expt(X,Y,W), mult(X,W,Z).

```

 ©: Michael Kohlhase 575 

Note: Viewed through the right glasses logic programming is very similar to functional programming; the only difference is that we are using $n+1$ -ary relations rather than n -ary functions. To see

how this works let us consider the addition function/relation example above: instead of a binary function $+$ we program a ternary relation `add`, where relation `add(X, Y, Z)` means $X + Y = Z$. We start with the same defining equations for addition, rewriting them to relational style.

The first equation is straight-forward via our correspondence and we get the ProLog fact `add(X, zero, X)`. For the equation $X + s(Y) = s(X + Y)$ we have to work harder, the straight-forward relational translation `add(X, s(Y), s(X + Y))` is impossible, since we have only partially replaced the function $+$ with the relation `add`. Here we take refuge in a very simple trick that we can always do in logic (and mathematics of course): we introduce a new name Z for the offending expression $X + Y$ (using a variable) so that we get the fact `add(X, s(Y), s(Z))`. Of course this is not universally true (remember that this fact would say that “ $X + s(Y) = s(Z)$ for all X, Y , and Z ”), so we have to extend it to a ProLog rule `add(X, s(Y), s(Z)) :- add(X, Y, Z)`. which relativizes to mean “ $X + s(Y) = s(Z)$ for all X, Y , and Z with $X + Y = Z$ ”.

Indeed the rule implements addition as a recursive predicate, we can see that the recursion relation is terminating, since the left hand sides are have one more constructor for the successor function. The examples for multiplication and exponentiation can be developed analogously, but we have to use the naming trick twice.

We now apply the same principle of recursive programming with predicates to other examples to reinforce our intuitions about the principles.

More Examples from elementary Arithmetics

- ▷ **Example 17.2.3** We can also use the `add` relation for subtraction without changing the implementation. We just use variables in the “input positions” and ground terms in the other two (possibly very inefficient since “generate-and-test approach”)

```
?-add(s(zero), X, s(s(s(zero)))) .
X = s(s(zero))
Yes
```

- ▷ **Example 17.2.4** Computing the the n^{th} Fibonacci Number (0, 1, 1, 2, 3, 5, 8, 13, ...; add the last two to get the next), using the addition predicate above.

```
fib(zero, zero) .
fib(s(zero), s(zero)) .
fib(s(s(X)), Y) :- fib(s(X), Z), fib(X, W), add(Z, W, Y) .
```

- ▷ **Example 17.2.5** using ProLog's internal arithmetic: a goal of the form `?- D is e.` where e is a ground arithmetic expression binds D to the result of evaluating e .

```
fib(0, 0) .
fib(1, 1) .
fib(X, Y) :- D is X - 1, E is X - 2, fib(D, Z), fib(E, W), Y is Z + W.
```



Note: Note that the `is` relation does not allow “generate-and-test” inversion as it insists on the right hand being ground. In our example above, this is not a problem, if we call the `fib` with the first (“input”) argument a ground term. Indeed, if match the last rule with a goal `?- g, Y.,` where g is a ground term, then $g-1$ and $g-2$ are ground and thus D and E are bound to the (ground) result terms. This makes the input arguments in the two recursive calls ground, and we get ground results for Z and W , which allows the last goal to succeed with a ground result for Y . Note as well that re-ordering the body literals of the rule so that the recursive calls are called before the computation literals will lead to failure.

We will now add the primitive data structure of lists to ProLog. They are very similar to lists in SML, only the concrete syntax is different. Just as in SML lists are constructed by prepending an element (the head) to an existing list (which becomes the rest list of the constructed one).

Adding Lists to ProLog

- ▷ Lists are represented by terms of the form `[a,b,c,...]`
- ▷ first/rest representation `[FR]`—, where `R` is a rest list.
- ▷ predicates for `member`, `append` and `reverse` of lists in default ProLog representation.

```
member(X, [X|_]).
member(X, [_|R]) :- member(X,R).
append([], L, L).
append([X|R], L, [X|S]) :- append(R,L,S).
reverse([], []).
reverse([X|R], L) :- reverse(R,S), append(S, [X], L).
```



©: Michael Kohlhase

577



Just as in SML, we can define list operations by recursion, only that we program with relations instead of with functions.

Logic programming is the third large programming paradigm (together with functional programming and imperative programming).

Relational Programming Techniques

- ▷ Parameters have no unique direction “in” or “out”

```
:- rev(L, [1,2,3]).
:- rev([1,2,3], L1).
:- rev([1,X], [2,Y]).
```

- ▷ Symbolic programming by structural induction

```
rev([], []).
rev([X,Xs], Ys) :- ...
```

- ▷ **Example 17.2.6** Generate and test

```
sort(Xs, Ys) :- perm(Xs, Ys), ordered(Ys).
```



©: Michael Kohlhase

578



From a programming practice point of view it is probably best understood as “relational programming” in analogy to functional programming, with which it shares a focus on recursion.

The major difference to functional programming is that relational programming does not have a fixed input/output distinction, which makes the control flow in functional programs very direct and predictable. Thanks to the underlying search procedure, we can sometime make use of the flexibility afforded by logic programming.

If the problem solution involves search (and depth-first search is sufficient), we can just get by with specifying the problem and letting the ProLog interpreter do the rest. In Example 17.2.6 we just specify that list `Xs` can be sorted into `Ys`, iff `Ys` is a permutation of `Xs` and `Ys` is ordered.

Given a concrete (input) list Xs , the ProLog interpreter will generate all permutations of Ys of Xs via the predicate `perm/2` and then test them whether they are ordered.

This is a paradigmatic example of logic programming. We can (sometimes) directly use the specification of a problem as a program. This makes the argument for the correctness of the program immediate, but may make the program execution non-optimal.

It is easy to see that the runtime of the ProLog program from Example 17.2.6 is not $O(n \log_2(n))$ which is optimal for sorting algorithms. This is the flip-side of the flexibility in logic programming. But ProLog has ways of dealing with that: the **cut operator**, which is a ProLog atom, which always succeeds, but which cannot be backtracked over. This can be used to prune the search tree in ProLog. We will not go into that here but refer the readers to the literature.

Now that we understand ProLog as a programming language, we will see that we already know most of the mechanics of the ProLog interpreter and that we can understand it as an instance of automated (resolution) theorem proving.

17.3 Logic Programming as Resolution Theorem Proving

To understand ProLog better, we can interpret the language of ProLog as resolution in PLNQ.

We know all this already

- ▷ Goals, goal-sets, rules, and facts are just clauses. (called "Horn clauses")
- ▷ **Observation 17.3.1** Rule $.H:-B_1, \dots, B_n$ corresponds to $H \vee \neg B_1 \vee \dots \vee \neg B_n$ (head the only positive literal)
- ▷ **Observation 17.3.2** (goal set) $?-G_1, \dots, G_n$ corresponds to $\neg G_1 \vee \dots \vee \neg G_n$
- ▷ **Observation 17.3.3** (fact) F corresponds to the unit clause F .
- ▷ **Definition 17.3.4** A **Horn clause** is a clause with at most one positive literal.
- ▷ **Note:** backchaining becomes (hyper)-resolution (special case for rule with facts) $\frac{P^t \vee A \quad P^f \vee B}{A \vee B}$ positive, unit-resulting hyperresolution (PURR)



This observation helps us understand ProLog better, and use implementation techniques from theorem proving.

PROLOG (Horn clauses)

- ▷ **Definition 17.3.5** Each clause contains at most one positive literal
 - ▷ $B_1 \vee \dots \vee B_n \vee \neg A$ ($A:-B_1, \dots, B_n$)
 - ▷ **Rule clause:** `fallible(X) :- human(X).`
 - ▷ **Fact clause:** `human(socrates).`
 - ▷ **Program:** set of rule and fact clauses
 - ▷ **Query:** `?- fallible(X), greek(X).`



PROLOG: Our Example

▷ Program:

```
human(sokrates).
human(leibniz).
greek(sokrates).
fallible(X) :- human(X).
```

▷ Example 17.3.6 (Query) ?- fallible(X),greek(X).

▷ Answer substitution: [sokrates/X]



©:Michael Kohlhase

581



Finally, we take a small detour and look at the role of inference (drawing conclusions from knowledge) to see that the deductive method is just one of three modes of inference.

Three Principal Modes of Inference

▷ Deduction: knowledge extension

$$\frac{\text{rains} \Rightarrow \text{wet_street} \quad \text{rains}}{\text{wet_street}} D$$

▷ Abduction explanation

$$\frac{\text{rains} \Rightarrow \text{wet_street} \quad \text{wet_street}}{\text{rains}} A$$

▷ Induction learning rules

$$\frac{\text{wet_street} \quad \text{rains}}{\text{rains} \Rightarrow \text{wet_street}} I$$



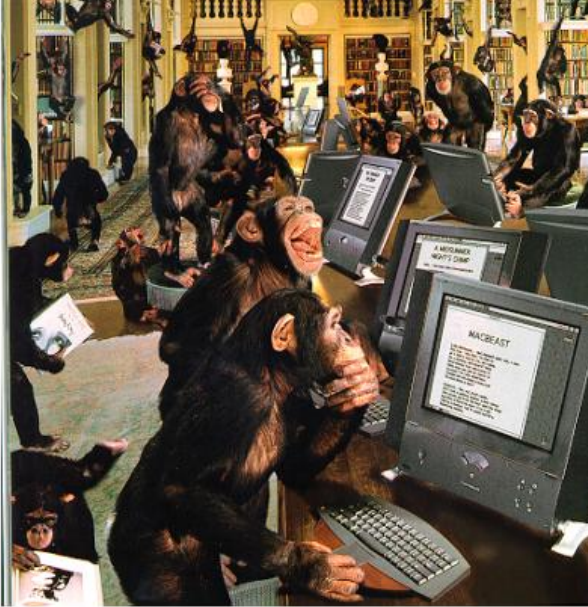
©:Michael Kohlhase


582







Part V

**A look back; What have we
learned?**





Let's hack!  \rightsquigarrow 2am in the CLAMV cluster

 ©: Michael Kohlhase 583 

 no, let's think 



- ▷ GIGO: Garbage In, Garbage Out (– ca. 1967)
- ▷ Applets, Not Craplets™ (– ca. 1997)

 ©: Michael Kohlhase 584 

What have we thought about in this year? We talked about various forms of

Machines Models
Algorithms, Languages and Programs
Information/Data/Representations

and their relation to each other

(and of course Math!)  ©: Michael Kohlhase 585 

Machine Models

- ▷ Abstract Interpreters (mind games)
- ▷ The SML interpreter/compiler (didn't we love recursive programming?)
- ▷ Combinatory Circuits, the register machine (this one could in theory solder)
- ▷ A stack based virtual machine (on top of the register machine)
- ▷ depth-first search as a model of computation in a knowledge base



©: Michael Kohlhase

586



Algorithms Languages and Programs

- ▷ Abstract data types (defining equations as recursive programs)
- ▷ standard ML (SML) (concrete ADTs with strong types, HO functions)
- ▷ elementary complexity analysis (Ooooooh, how fast this class grows)
- ▷ Assembler language on the register machine (finally, an imperative language)
- ▷ Turing Machines, universality and the halting problem (mind games again)
- ▷ concrete algorithms: search (informed and uninformed)
- ▷ logic programming in PROLOG (a crash course in declarative programming)



©: Michael Kohlhase

587



Information, Data, and Representations

- ▷ term representations and substitutions (constants, variables, function application)
- ▷ Codes as transformations on formal languages. (programs as a special case)
- ▷ Boolean Expressions and Boolean functions (syntax and semantics)
- ▷ Hilbert, Resolution, Tableau, calculi (correct, complete)
- ▷ positional number systems (in particular binary numbers)



©: Michael Kohlhase

588



The (Discrete) Math involved

- ▷ sets, relations, functions,
- ▷ natural numbers and proof by induction (such a lot of talk about something so simple)
- ▷ the axiomatic/deductive method in Math (play the math game by the rules)

- ▷ formal languages and codes (are just sets of strings and injective mappings)
- ▷ Boolean algebra, axioms, deduction, prime implicants
- ▷ elementary Graph theory (and the algorithms on that)



©:Michael Kohlhase

589



Search Algorithms as a Classes of dedicated Algorithms

- ▷ uninformed search (only the information in the problem)
- ▷ informed search (additionally heuristics as a sense of direction)
- ▷ local search (when we are not interested in the solution path)



©:Michael Kohlhase

590



The Internet and World-Wide Web

- ▷ The Internet as a packet-switched Network of networks. . .
- ▷ . . . driven by special protocols (Ethernet, TCP/IP, SMTP, . . .)
- ▷ Security by encryption.
- ▷ Five components of the WWWeb: URI, Web server, Browser, HTTP, HTML.
- ▷ XML as a general tree communication infrastructure



©:Michael Kohlhase

591



Bibliography

- [AES01] Announcing the ADVANCED ENCRYPTION STANDARD (AES), 2001.
- [BCHL09] Bert Bos, Tantek Celik, Ian Hickson, and Høakon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C Candidate Recommendation, World Wide Web Consortium (W3C), 2009.
- [BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force (IETF), 2005.
- [Chu36] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, pages 40–41, May 1936.
- [Con12] Jamie Condliffe. Easily understand encryption using...paint and clocks? *Gizmodo*, 2012.
- [Den00] Peter Denning. Computer science: The discipline. In A. Ralston and D. Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 405–419. Nature Publishing Group, 2000.
- [DH98] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, Internet Engineering Task Force (IETF), 1998.
- [Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [ECM09] ECMAScript language specification, December 2009. 5th Edition.
- [FGM⁺99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force (IETF), 1999.
- [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.
- [HASB13] Ivan Herman, Ben Adida, Manu Sporny, and Mark Birbeck. RDFa 1.1 primer – second edition. W3C Working Group Note, World Wide Web Consortium (W3C), 2013.
- [HL11] Martin Hilbert and Priscila López. The world’s technological capacity to store, communicate, and compute information. *Science*, 331, feb 2011.
- [Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
- [KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C recommendation, World Wide Web Consortium (W3C), 2004.

- [Koh06] Michael Kohlhase. *OMDOC – An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006.
- [Koh08] Michael Kohlhase. Using \LaTeX as a semantic markup format. *Mathematics in Computer Science*, 2(2):279–304, 2008.
- [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.
- [Koh15] Michael Kohlhase. \sTeX : Semantic markup in \TeX / \LaTeX . Technical report, Comprehensive \TeX Archive Network (CTAN), 2015.
- [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.
- [OWL09] OWL Working Group. OWL 2 web ontology language: Document overview. W3C recommendation, World Wide Web Consortium (W3C), October 2009.
- [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.
- [RFC80] DOD standard internet protocol, 1980.
- [RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 Specification. W3C Recommendation REC-html40, World Wide Web Consortium (W3C), April 1998.
- [RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence — A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.
- [Ros90] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 1990.
- [Rya] Konstantin Ryabitsev. PGP Web of Trust: Core concepts behind trusted communication. <http://www.linux.com/learn/tutorials/760909-pgp-web-of-trust-core-concepts>. seen 2014-09-20.
- [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.
- [XAM] apache friends - xampp. <http://www.apachefriends.org/en/xampp.html>.
- [Zus36] Konrad Zuse. Verfahren zur selbsttätigen durchführung von rechnungen mit hilfe von rechenmaschinen. Patent Application Z23139/GMD Nr. 005/021, 1936.

Index

- λ -notation, 40
- \mathcal{C} -derivation, 127
- k -th intermediate carry, 203
- n -ary gates, 185
- n -bit
 - full
 - adder, 194
- n -dim
 - Cartesian
 - space, 36
- n -fold
 - Cartesian
 - product, 36
- n -tuple, 36
-)
 - backtracking, 362
- A^* search, 346
- HTTP
 - Secure, 301
- Secure
 - HTTP, 301
- Peano Axioms for $\mathcal{L}[\mathbb{N}]$, 55
- ProLog program, 358
- ProLog terms, 358
- Abduction, 367
- abstract
 - call-by-value
 - interpreter, 73
 - computation, 65
 - data
 - type, 62
 - interpreter, 65
 - procedure, 65, 71
 - program, 72
- access
 - random (memory), 216
- accumulator, 216
- action
 - table, 253
- active, 7
- acyclic, 178
- adder, 192
 - carry chain, 194
 - conditional sum, 196
 - full, 193
 - half, 192
- addition
 - carry chain, 191
 - operation, 29
 - rules, 191
- address
 - decoder, 210
 - IPv4, 266
 - IPv6, 266
 - MAC, 266
- admissible, 127, 346
- admits
 - weakening, 126
- advanced
 - query
 - operator, 286
- agent
 - user, 276
- algebra
 - Boolean, 101
- Algorithm
 - Union Find, 17
- algorithm, 16
 - generate-and-test, 114
 - search, 319
- alike
 - share, 161
- alphabet, 87
- ALU, 206, 215–218
- American Standard Code for Information Interchange, 93
- an exception
 - Raising, 81
- anchor
 - cell, 245
- anonymous
 - variables, 46
- answer
 - substitution, 360
- Anti-Counterfeiting Trade Agreement, 155
- antisymmetric, 38
- append
 - function, 56
- application, 68

- layer, 269
 - web, 282
 - web (framework), 282
- Architectural works, 156
- argument, 66
 - sort, 65
- arithmetic, 124
 - logic
 - unit, 206
- assembler, 221
 - language, 216–218
- assignment
 - variable, 100
- assumption, 126
- asymmetric, 38
- asymmetric-key
 - cryptosystem, 296
- asymptotically
 - bounded, 106
- asynchronous, 209
- atom, 121, 141
- atomic, 141
- attribution, 161
- attribute, 308
 - node, 308
- Audiovisual works, 156
- authority
 - certificate, 300
- Axiom
 - Peano, 63
- axiom, 26, 127
- axioms
 - Peano, 25
- backtrack
 - point, 362
- backtracking
 -), 362
- balanced, 183
 - bracketing
 - structure, 308
- base, 189
 - case, 28
 - condition, 25, 28
 - equation, 30
 - knowledge, 358
 - sort, 62
- basic
 - multilingual
 - plane, 95
 - operator, 45
 - type, 44
 - constructor, 44
- Berne
 - convention, 155
- bijection, 190
- bijective, 41
- binary, 89
 - natural
 - number, 199
 - tree, 183
 - unit
 - prefix, 213
- bit, 212
 - carry, 192
 - most significant, 199
 - sign, 199
 - sum, 192
- Blaise Pascal, 124
- body, 83
- Boolean
 - algebra, 101
 - expressions, 181
 - function, 103
 - polynomial, 104
 - sum, 100
- bootstrapping
 - process, 190
- borrow
 - input (bit), 202
- bounded
 - asymptotically, 106
- bracketing
 - balanced (structure), 308
- breaking
 - code, 290
- Browser
 - web, 275
- browsing, 273
- by description
 - definition, 33
- byte, 212
 - code, 221, 232
- calculus, 127
 - resolution, 148
- call-by-value
 - abstract (interpreter), 73
- canonical
 - product, 104
 - sum, 104
- card
 - punch, 93
- cardinality, 41
- carry
 - bit, 192
 - input, 193
- carry chain

- adder, 194
- addition, 191
- Cartesian, 50
 - n -dim (space), 36
 - n -fold (product), 36
 - product, 36
- cascaded, 50
- Cascading Style Sheets, 279
- case
 - base, 28
 - step, 28
- cell
 - anchor, 245
 - internal, 245
- certificate
 - authority, 300
- chain
 - infinite, 75
- character, 87
 - code, 89, 190
 - encoding, 95
 - stop, 92
 - structural, 96
- cheating
 - passive, 7
- child, 179
- Choreographic works, 155
- cipher, 290
- ciphertext, 290
- circuit
 - combinational, 180
- civil
 - law
 - tradition, 154
- clause, 104, 148, 358
 - empty, 148
 - Horn, 366
 - set, 104
- clock, 209
- closed, 67, 142
- closing
 - tag, 308
- CNF, 104
- CNF transformation calculus, 149
- code
 - breaking, 290
 - byte, 221, 232
 - character, 89, 190
 - Morse, 90
 - on strings, 91
 - point, 95
 - prefix, 90
 - string, 91, 190
- codeword, 89
- codomain, 39
- combinational
 - circuit, 180
- combinational circuit
 - with constants, 181
- with constants
 - combinational circuit, 181
- command interpreter, 224
- comment, 45
- commercial
 - use, 161
- common
 - law
 - tradition, 154
- communication, 80
- comparable, 39
- compiler, 221
- complete, 128
 - Turing, 255
- Completeness, 135
- complex, 141
- complexity
 - theory, 18
- composition, 38, 42
- comprehension
 - set, 35
- computation, 65
 - abstract, 65
- Computer
 - Science, 1
- computer-created, 285
- computes, 182
- concatenation, 87, 88
- conclusion, 126
- condition
 - base, 25, 28
 - step, 25, 28, 29
- conditional sum
 - adder, 196
- congruent to b modulo, 294
- conjecture, 26
- conjunctive
 - normal
 - form, 104, 148
- constant
 - name, 96
- constructive
 - proof, 108
- constructor, 47, 54
 - declaration, 57, 62
 - ground (term), 63
 - term, 67
- control, 197
 - structure, 45

- convention
 - Berne, 155
- converse, 38
- cookie, 283
- cookies
 - third party, 283
- copyleft, 160
- copyright, 157
 - holder, 157
 - infringement, 157
- copyrightable
 - work, 155
- copyrighted, 156
- corollary, 26
- correct, 128
- Correctness, 135
- cost, 105, 182
- countable, 41
- countably
 - infinite, 41
- counter
 - program, 216, 217
- counterexamples, 123
- CPU, 217
- crawler
 - web, 285
- Creative Commons
 - license, 161
- cryptoanalysis, 290
- cryptography, 290
- cryptology, 290
- cryptosystem
 - asymmetric-key, 296
 - public-key, 296
- current
 - instruction, 217
 - state, 351
- Currying, 49
- cut
 - operator, 366
- cycle, 178
- cyclic, 178
- D-flipflop, 208
- DAG, 178
- data, 209
 - abstract (type), 62
 - store, 216
 - type, 57
- declaration
 - constructor, 57, 62
 - function, 45
 - namespace, 308
 - procedure, 70
 - type, 45
 - value, 45
- decoder
 - address, 210
- decryption, 290
- Deduction, 367
- defined
 - inductively, 47
 - inductively (set), 54
- definiendum, 33
- definiens, 33
- defining equations for substitution application, 69
- definition
 - by description, 33
 - implicit, 33
 - simple, 33
- depth, 67, 99, 178, 179
 - first
 - search, 334
- derivation, 26
 - relation, 126
- derivative
 - works, 161
- derived
 - inference
 - rule, 146, 149
 - rule, 145
- derives, 143
- difference
 - set, 36
- digit, 189, 190
- digital
 - signature, 297
- digraph, 174
- directed
 - edge, 174
 - graph, 174, 175
- discharged, 136
- discrete
 - logarithm, 294
- disjunctive
 - normal
 - form, 104, 148
- diverges, 75
- DNF, 104, 148, 186
- DNS, 269
- document
 - object
 - model, 283, 308
 - root, 308
 - XML (tree), 308
- Document markup, 277
- domain, 39

- public, 156
- Domain Name System, 269
- dominates, 111, 350
- door
 - trap (function), 296
- Dramatic works, 155
- edge, 175
 - directed, 174
 - undirected, 174
- edges, 14
- element
 - empty, 308
 - node, 308
- empty
 - clause, 148
 - element, 308
 - set, 36
 - string, 87
- enable input, 209
- encoding
 - character, 95
- Encryption, 290
- end, 177
- end-user
 - license
 - agreement, 159
- entails, 122
- environment, 232
- equality
 - set, 35
- equation
 - base, 30
 - step, 30
- equivalence
 - relation, 38
- equivalent, 101, 103, 176
- essential, 115
- evaluation
 - function, 100, 346
- exbi, 213
- exception, 81
 - handler, 83
- exceptions, 80
- exclusive
 - or, 187
- exploitation
 - rights, 157
- exponentiation
 - unary, 31
- expression, 97, 229, 235
 - label, 182
- expressions
 - Boolean, 181
 - extension, 90
- fact, 358
- Fact clause, 366
- fair
 - use
 - doctrine, 158
- false under φ , 122
- falsifiable, 122
- Fibonacci
 - number, 77
 - sequence, 77
- final
 - state, 253
- finite, 41
- firmware, 218
- first
 - depth (search), 334
- first-order
 - natural deduction
 - calculus, 138
- first-order logic with equality, 138
- folding
 - left (operator), 50
 - right (operator), 52
- formal
 - language, 88
 - system, 126, 127
- formula
 - labeled, 141
- formulation of
 - Propositional Logic, 121
- frame
 - pointer, 245
- Free/Libre/Open-Source
 - Software, 160
- fringe, 326
- full
 - n -bit (adder), 194
 - adder, 193
- fully balanced, 183
- function
 - append, 56
 - Boolean, 103
 - declaration, 45
 - evaluation, 100, 346
 - identity, 40
 - interpretation, 100
 - inverse, 41
 - name, 96
 - named, 235
 - one-way, 297
 - partial, 39
 - result, 75

- selector, 46
- sort, 62
- space, 40
- total, 39
- transition, 253
- trapdoor, 297
- functional, 43
 - programming
 - language, 218
 - variable, 235
- gate, 180
- General
 - Public
 - License, 160
- generate-and-test, 17
 - algorithm, 114
- gibi, 213
- goal
 - state, 320
- goals, 359
- Gottfried Wilhelm Leibniz, 124
- graph, 13
 - directed, 174, 175
 - isomorphism, 176
 - labeled, 177
 - undirected, 174, 175
- greedy
 - search, 343
- ground, 67
 - constructor
 - term, 63
- half
 - adder, 192
- halting
 - problem, 257
- handled, 83
- handler
 - exception, 83
- head, 63
- heuristic, 343
- hexadecimal
 - numbers, 217
- holder
 - copyright, 157
- Horn
 - clause, 366
- http
 - request, 276
- human-organized, 285
- hybrid, 285
- hyperlink, 272
- hypertext, 273
- HyperText Markup Language, 278
- Hypertext Transfer Protocol, 276
- hypotheses, 127
- hypothetical
 - reasoning, 136
- IANA, 266
- ICANN, 269
- idempotent, 276
- identity
 - function, 40
- idiom
 - math, 32
- IETF, 272
- image, 42
- immediate
 - subterm, 71
- imperative
 - programming
 - language, 218
- implicant, 111
 - prime, 112
- implicit
 - definition, 33
- implies, 111
- in-degree, 174
- index
 - register, 219
 - search, 287
- indexing, 287
- induced, 103
- induces, 71
- Induction, 367
- inductive, 47
- inductively
 - defined, 47
 - set, 54
- inference, 26
 - derived (rule), 146, 149
 - rule, 126, 148
- infinite
 - chain, 75
 - countably, 41
- infinite precision
 - integers, 78
- information
 - privacy, 161
- infringement
 - copyright, 157
- initial, 175, 176
 - state, 253, 320
- injective, 41
- input, 180
 - borrow

- bit, 202
- carry, 193
- instance, 69
- instantiates, 69
- instruction
 - current, 217
 - program, 218
- integers
 - infinite precision, 78
- intellectual
 - property, 153
- interface
 - network, 266
- internal
 - cell, 245
- Internet, 261
- Internet Assigned Numbers Authority, 266
- Internet Engineering Task Force, 272
- Internet Protocol, 266
- Internet Protocol Suite, 265
- Internet Protocol Version 4, 266
- Internet Protocol Version 6, 266
- interpretation
 - function, 100
- interpreter
 - abstract, 65
- intersection, 35
- intersection over a collection, 36
- invariants
 - loop, 220
- inverse
 - function, 41
- IP, 266
 - packet, 267
- IPv4, 266
 - address, 266
- IPv6, 266
 - address, 266
- irreflexive, 38
- ISO-Latin, 94
- isomorphism
 - graph, 176
- iterator, 51
- jump
 - table, 226
- Karnaugh-Veitch
 - map, 118
- key, 232, 287, 290
 - pair, 296
 - private, 296
 - public, 296
 - public (certificate), 300
- kibi, 213
- knowledge
 - base, 358
- KV-map, 118
- label, 177
 - expression, 182
- labeled
 - formula, 141
 - graph, 177
- Landau
 - set, 106
- language
 - assembler, 216–218
 - formal, 88
- law
 - civil (tradition), 154
 - common (tradition), 154
- layer
 - application, 269
 - transport, 268
- leaf, 179
- leak
 - memory, 234
- left
 - folding
 - operator, 50
- lemma, 26
- length, 178
- lexical
 - order, 89
- license, 158
 - Creative Commons, 161
 - end-user (agreement), 159
- licensee, 158
- licensor, 158
- LIFO, 224
- linear, 39
- list
 - variable, 103
- literal, 104, 141, 144
- Literary
 - work, 155
- local
 - search, 351
- logarithm
 - discrete, 294
- logic
 - arithmetic (unit), 206
 - sequential (circuit), 207
- logical
 - reasoning, 26
 - system, 26
- loop

- invariants, 220
- MAC
 - address, 266
- Machine
 - Turing, 253
- machine, 221
 - register, 216, 217
 - state, 79
 - Turing (specification), 253
 - virtual, 221
- management
 - memory (unit), 216
- map
 - Karnaugh-Veitch, 118
- matcher, 69
- matching
 - pattern, 46
- math
 - idiom, 32
- mathematical
 - vernacular, 32
- MathTalk, 32
- maze, 14
- mebi, 213
- media access control address, 266
- memory
 - leak, 234
 - management
 - unit, 216
 - random access, 216
- metasearch, 285
- microML, 235
- minimal
 - polynomial, 186
- MMU, 216–218
- model, 100
- monomial, 104, 148
- Morse
 - code, 90
- most significant
 - bit, 199
- multilingual
 - basic (plane), 95
- multiplexer, 196
- multiplication
 - unary, 31
- multiword queries, 286
- Musical works, 155
- mutation, 80
- name
 - constant, 96
 - function, 96
 - variable, 96
- named
 - function, 235
 - variable, 229
- namespace
 - declaration, 308
- natural
 - binary (number), 199
 - unary (numbers), 24
- natural deduction
 - first-order (calculus), 138
 - propositional (calculus), 136
- navigating, 273
- negative, 143
- network
 - interface, 266
 - packet-switched, 264
 - packets, 264
- network interface controller, 265
- NIC, 265
- node, 174, 175
 - attribute, 308
 - element, 308
 - root, 15
 - text, 308
- nodes, 14
- nonempty
 - string, 88
- normal
 - conjunctive (form), 104, 148
 - disjunctive (form), 104, 148
- number
 - Fibonacci, 77
 - positional (system), 189
- numbers
 - hexadecimal, 217
- object
 - document (model), 283, 308
- off
 - worked, 148
- offline
 - problem
 - solving, 319
- Offline problem solving, 319
- on
 - relation, 38
- on strings
 - code, 91
- one, 28
- one-way
 - function, 297
- Online problem solving, 319
- open, 142

- opening
 - tag, 308
- operation
 - addition, 29
- operator, 320
 - basic, 45
 - cut, 366
- or
 - exclusive, 187
- order
 - lexical, 89
 - partial, 39
- ordered
 - pair, 174, 175
- out-degree, 174
- output, 180, 229
- overflow, 205
- packet
 - IP, 267
- packet-switched
 - network, 264
- packets
 - network, 264
- page
 - web, 272
- pair, 36
 - key, 296
 - ordered, 174, 175
- parent, 179
- parse-tree, 180
- partial
 - function, 39
 - order, 39
 - strict (order), 39
- passive
 - cheating, 7
- path, 177
 - XML (language), 308
- pattern
 - matching, 46
- Peano
 - Axiom, 63
 - axioms, 25
- pebi, 213
- personal
 - rights, 157
- PGS
 - work, 156
- Pictorial, Graphic and Sculptural works, 156
- PIT, 115
- plaintext, 290
- point
 - backtrack, 362
 - code, 95
- pointer
 - frame, 245
 - stack, 225
- polarity, 199
- polynomial
 - Boolean, 104
 - minimal, 186
- port, 268
- positional
 - number
 - system, 189
- postulate, 26
- power
 - set, 36
- pre-image, 42
- predecessor, 25
- Predicate Logic without Quantifiers, 121
- prefix, 89
 - code, 90
 - proper, 89
- prime
 - implicant, 112
- privacy
 - information, 161
- private
 - key, 296
- problem
 - halting, 257
 - offline (solving), 319
- problem (formulation), 320
- procedure
 - abstract, 65, 71
 - declaration, 70
- process
 - bootstrapping, 190
- product, 100
 - canonical, 104
 - Cartesian, 36
 - sort, 62
 - term, 104
 - unary, 31
- product of
 - sums, 104
- products
 - sum of, 104
- Program, 366
- program
 - abstract, 72
 - counter, 216, 217
 - instruction, 218
 - store, 216
- program store, 224
- programming

- functional (language), 218
- imperative (language), 218
- proof, 26, 127
 - constructive, 108
 - tableau, 143
- proof-reflexive, 126
- proof-transitive, 126
- proper
 - prefix, 89
- proper subset, 35
- proper superset, 35
- property
 - intellectual, 153
- proposition, 121
- propositional
 - natural deduction
 - calculus, 136
- Propositional Logic
 - formulation of, 121
- Public
 - General (License), 160
- public
 - domain, 156
 - key, 296
 - certificate, 300
- public-key
 - cryptosystem, 296
- pull-back, 191, 192
- punch
 - card, 93
- Query, 366
- query, 359
 - advanced (operator), 286
- Quine-McCluskey, 186
- radix, 189
- Raising
 - an exception, 81
- RAM, 216, 217
- random
 - access
 - memory, 216
- random access
 - memory, 216
- realizes, 103
- reasoning
 - hypothetical, 136
 - logical, 26
- recipe, 16
- recursion
 - relation, 75
 - step, 74
- recursive, 47, 75
- reflexive, 38
- refutation
 - resolution, 149
 - tableau, 143
- register, 215–217
 - index, 219
 - machine, 216, 217
 - state, 253
- relation, 38, 175
 - derivation, 126
 - equivalence, 38
 - on, 38
 - recursion, 75
- relative
 - URI, 273
- renewal, 159
- representation, 13
- request
 - http, 276
- Request for Comments, 272
- Reset, 208
- resolution
 - calculus, 148
 - refutation, 149
 - sproof, 149
- resolve, 117
- resolvent, 113
- resource
 - uniform (identifier), 273
 - uniform (locator), 274
 - uniform (name), 274
 - web, 273
- restriction, 42
- result, 66
 - function, 75
 - sort, 65
- RFC, 272
- Ridge, 353
- right
 - folding
 - operator, 52
- rights
 - exploitation, 157
 - personal, 157
- root, 179
 - document, 308
 - node, 15
- rosary, 16
- router, 265
- RS-flipflop, 208
- RS-latch, 208
- RTFM, 84
- rule, 64, 71, 358
 - derived, 145

- inference, 126, 148
- Rule clause, 366
- rules
 - addition, 191
- safe, 276
- satisfiable, 122
- saturated, 142
- Science
 - Computer, 1
- scripting
 - server-side (framework), 280
 - server-side (language), 280
- scytale, 290
- search
 - algorithm, 319
 - greedy, 343
 - index, 287
 - local, 351
 - strategy, 326
 - tree (algorithm), 324
 - web (engine), 285
 - web (query), 286
- selector
 - function, 46
- semantic
 - web, 309
- semantics, 97
- semi-prime, 298
- sequence
 - Fibonacci, 77
- sequential
 - logic
 - circuit, 207
- server
 - web, 276
- server-side
 - scripting
 - framework, 280
 - language, 280
- set, 174
 - clause, 104
 - comprehension, 35
 - difference, 36
 - empty, 36
 - equality, 35
 - Landau, 106
 - power, 36
- Set inputs, 208
- share
 - alike, 161
- sign
 - bit, 199
- signature, 70
 - digital, 297
- simple, 178
 - definition, 33
- simple while language, 229
- sink, 176
- site
 - web, 272
- size, 36
- Software
 - Free/Libre/Open-Source, 160
- solution, 321
- sort
 - argument, 65
 - base, 62
 - function, 62
 - product, 62
 - result, 65
- sorts, 62
- sound, 128
- Sound recording, 155
- source, 176
- space
 - function, 40
- spanning
 - tree, 15
- spider, 285
- sproof
 - resolution, 149
- stack, 224
 - pointer, 225
- Standard
 - Unicode, 95
- start, 177
 - value, 51
- state, 253, 320
 - current, 351
 - final, 253
 - goal, 320
 - initial, 253, 320
 - machine, 79
 - register, 253
- statement, 229
- step
 - case, 28
 - condition, 25, 28, 29
 - equation, 30
 - recursion, 74
- stop
 - character, 92
- store
 - data, 216
 - program, 216
- strategy
 - search, 326

- strict
 - partial
 - order, 39
- String, 48
- string, 87, 88
 - code, 91, 190
 - empty, 87
 - nonempty, 88
- structural
 - character, 96
- structure
 - control, 45
- sub-monomial, 111
- subset, 35
- substitution, 68
 - answer, 360
- substring, 88
- subterm, 71
 - immediate, 71
- subtractor, 202
- successor, 25
- sum
 - bit, 192
 - Boolean, 100
 - canonical, 104
 - term, 104
- sum of
 - products, 104
- summation
 - unary, 31
- sums
 - product of, 104
- superset, 35
- support, 68
- surjective, 41
- symmetric, 38
- Symmetric-key cryptosystems, 291
- synchronous, 209
- syntax, 97
- system
 - formal, 126, 127
 - logical, 26
- \mathcal{T}_0 -theorem, 142
- table
 - action, 253
 - jump, 226
- tableau
 - proof, 143
 - refutation, 143
- tag
 - closing, 308
 - opening, 308
- tape, 253
- TCP/IP, 265
- tebi, 213
- term, 70, 159
 - constructor, 67
 - product, 104
 - sum, 104
- terminal, 176
- terminates, 66, 73
- terminates, 75
- territory, 159
- test calculi, 143
- text
 - node, 308
- theorem, 26, 127
- theory
 - complexity, 18
- third party
 - cookies, 283
- TLD, 269
- TLS
 - connection, 301
- connection
 - TLS, 301
- TLS handshake, 301
- top-level domain, 269
- total, 38
 - function, 39
- transition
 - function, 253
- transitive, 38
- Transmission Control Protocol, 268
- transport
 - layer, 268
- Transport layer security, 300
- trap
 - door
 - function, 296
- trapdoor, 297
 - function, 297
- tree, 179
 - binary, 183
 - search
 - algorithm, 324
 - spanning, 15
- true under φ , 122
- Turing
 - complete, 255
 - Machine, 253
 - machine
 - specification, 253
 - universal (machine), 256
- twelve, 28
- two, 28
- two's complement, 200

- two's complement adder, 202
- type, 45
 - basic, 44
 - basic (constructor), 44
 - data, 57
 - declaration, 45
 - universal, 46
- unary
 - exponentiation, 31
 - multiplication, 31
 - natural
 - numbers, 24
 - product, 31
 - summation, 31
- undefined at, 39
- underflow, 205
- undirected
 - edge, 174
 - graph, 174, 175
- Unicode
 - Standard, 95
- uniform
 - resource
 - identifier, 273
 - locator, 274
 - name, 274
- union, 35
- Union Find
 - Algorithm, 17
- union over a collection, 35
- Union-Find, 17
- unit
 - binary (prefix), 213
- universal
 - Turing
 - machine, 256
 - type, 46
- Universal Character Set, 95
- universe, 100
- unsatisfiable, 122, 148
- URI, 273
 - relative, 273
- URL, 274
- URN, 274
- use
 - commercial, 161
 - fair (doctrine), 158
- user
 - agent, 276
- User Datagram Protocol, 268
- UTM, 256
- valid, 122, 303
- value
 - declaration, 45
 - start, 51
- variable, 67
 - assignment, 100
 - functional, 235
 - list, 103
 - name, 96
 - named, 229
- variables
 - anonymous, 46
- vector, 36
- vernacular
 - mathematical, 32
- vertices, 174
- virtual
 - machine, 221
- virtual program counter, 224
- visualization, 15
- VM Arithmetic Commands, 222
- control
 - operator, 222
- operator
 - control, 222
- imperative access to variables, 223
- VPC, 224
- weakening
 - admits, 126
- web
 - application, 282
 - framework, 282
 - Browser, 275
 - crawler, 285
 - page, 272
 - resource, 273
 - search
 - engine, 285
 - query, 286
 - semantic, 309
 - server, 276
 - site, 272
- wildcard, 286
- Wilhelm Schickard, 124
- word, 216, 221
- work
 - copyrightable, 155
 - Literary, 155
 - PGS, 156
- work made for hire, 157
- worked
 - off, 148
- works
 - derivative, 161

World Wide Web, 261
WWW, 261

XML
 document
 tree, 308
 path
 language, 308

yobi, 213

zebi, 213
zero, 25, 27