

General Computer Science

320101 — Fall 2016

Michael Kohlhase

Computer Science, Jacobs University Bremen, Germany
<http://kwarc.info>

December 8, 2016

Chapter 1 Getting Started with “General Computer Science”

1.1 Overview over the Course

Plot of “General Computer Science”

- ▶ Today: Motivation, Admin, and find out what you already know
 - ▶ What is Computer Science?
 - ▶ Information, Data, Computation, Machines
 - ▶ a (very) quick walk through the topics
- ▶ Get a feeling for the math involved (⚠ not a programming course!!! ⚠) (so we can talk rigorously)
 - ▶ learn mathematical language
 - ▶ 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)
- ▶ Representing knowledge in formal languages and reasoning about them
 - ▶ formal languages and their operations
 - ▶ syntax vs. semantics (form vs. function)
 - ▶ inferenced systems for understanding (logical) argumenation

1.2 Administrativa

1.2.1 Grades, Credits, Retaking

Prerequisites, Requirements, Grades

- ▶ **Prerequisites:** Motivation, Interest, Curiosity, hard work

- ▶ You can do this course if you want!

- ▶ **Grades:** (plan your work involvement carefully)

Tuesday 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.

- ▶ **Tuesday Quizzes:** (Almost) every tuesday, 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.)

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)

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.

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)



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)

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

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

1.2.3 Resources

Textbooks, Handouts and Information, Forum

- ▶ **No required textbook, but course notes, posted slides**
- ▶ Information resources (e.g. Course notes) will be posted at <http://kwarc.info/teaching/GenCS>
- ▶ 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)
 - ▶ **Set Up PantaRhei Access:** to get notifications
 - 1) Log into <http://panta.kwarc.info>, (use your Jacobs login)
 - 2) find the course “GenCS Fall2016”, (this course)
 - 3) request membership (I will approve you)
 - ▶ if there are problems send e-mail to course-gencs-tas@jacobs-university.de

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, Mac OS X, cygwin) (prevalent in CS)
 - ▶ FireFox (just a better browser (for Math))
- ▶ learn how to touch-type NOW (reap the benefits earlier, not later)

Chapter 2 Motivation and Introduction

2.1 What is Computer Science?

What is Computer Science about?

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

An Answer:

Let's hack

2am in the IRC Quiet Study Area



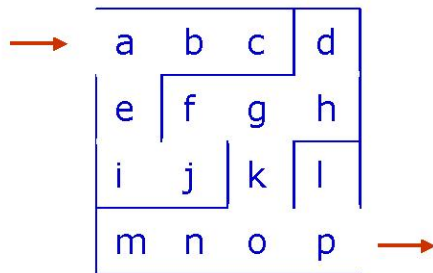
⚠ no, let's think ⚠

- ▶ “*The GIGO Principle: Garbage In, Garbage Out*” (– ca. 1967)
- ▶ “*Applets, Not Crapletstm*” (– ca. 1997)

2.2 Computer Science by Example

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.



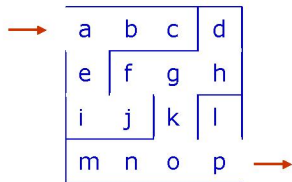
- ▶ **Mathematical Formulation:**
 - ▶ a set of rooms: $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$
 - ▶ Pairs of adjacent rooms that have an open wall between them.
- ▶ **Example 2.1** For example, (a, b) and (g, k) are pairs.
- ▶ Abstractly speaking, this is a mathematical structure called a **graph**.

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”.

Mazes as Graphs

- ▶ **Definition 2.2** Informally, a graph consists of a set of **nodes** and a set of **edges**.
(a good part of CS is about graph algorithms)
- ▶ **Definition 2.3** 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

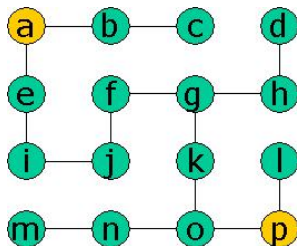
$$\left\langle \left\{ \begin{array}{l} (a, e), (e, i), (i, j), \\ (f, j), (f, g), (g, h), \\ (d, h), (g, k), (a, b) \\ (m, n), (n, o), (b, c) \\ (k, o), (o, p), (l, p) \end{array} \right\}, a, p \right\rangle$$

Mazes as Graphs (Visualizing Graphs via Diagrams)

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

Our maze

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

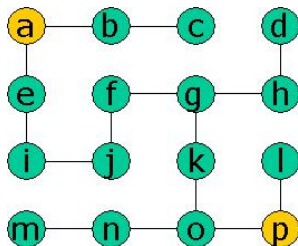


can be visualized as

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

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.



Mazes as trees

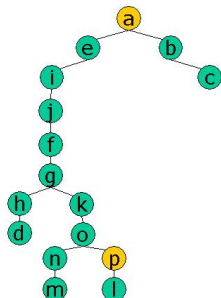
▶ **Definition 2.4** Informally, a tree is a graph:

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

▶ **Definition 2.5** A **spanning tree** is a tree that includes all of the nodes.

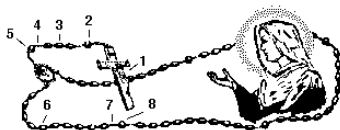
Q: Why is it good to have a spanning tree?

- ▶ A: Trees have no cycles! (needed for uniqueness)
- ▶ A: Every room is reachable from the root!



Algorithm

- ▶ Now that we have a data structure in mind, we can think about the algorithm.
- ▶ **Definition 2.6** An **algorithm** is a series of instructions to control a (computation) process



- ▶ **Example 2.7 (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.

Creating a spanning tree

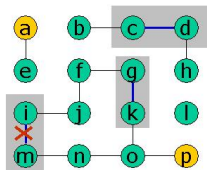
- ▶ When adding a wall to the tree, how do we detect that it won't create a cycle?
- ▶ When adding wall (x, y) , we want to know if there is already a path from x to y in the tree.
- ▶ In fact, there is a fast algorithm for doing exactly this, called "Union-Find".

Definition 2.8 (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 (x, y) to the tree, it makes sure that x and y are not in the same equivalence class.

Example 2.9 A partially constructed maze



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**.

Performance and Scaling

- ▶ Suppose we have three algorithms to choose from. (which one to select)
- ▶ Systematic analysis reveals performance characteristics.
- ▶ **Example 2.10** 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	$100\mu s$	$7\mu s$	$2\mu s$
5	$.5ms$	$175\mu s$	$32\mu s$
10	$1ms$	$.7ms$	$1ms$
45	$4.5ms$	$14ms$	$1.1Y$
100
1 000
10 000
1 000 000

What?! One year?

- ▶ $2^{10} = 1\,024$ ($\approx 1024\mu\text{s}, 1\text{ms}$)
- ▶ $2^{45} = 35\,184\,372\,088\,832$ ($3.5 \times 10^{13}\mu\text{s} = 3.5 \times 10^7\text{s} \sim 1.1\text{Y}$)
- ▶ **Example 2.11** we denote all times that are longer than the age of the universe with –

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

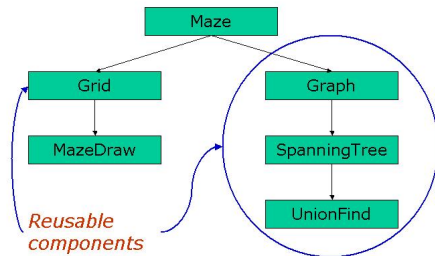
2.3 Other Topics in Computer Science

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.



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.

Part I Representation and Computation

Chapter 3 Elementary Discrete Math

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].

3.1 Mathematical Foundations: Natural Numbers

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)

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)



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 1.5** For $////$, Apply the o -rule once and then the s -rule four times.
- ▶ **Definition 1.6** we call these representations **unary natural numbers**.

A little more sophistication (math) please

- ▶ **Definition 1.7** We call a unary natural number the **successor** (predecessor) of another, if it can be constructed by adding (removing) a slash. (successors are created by the **s-rule**)
- ▶ **Example 1.8** $///$ is the successor of $//$ and $//$ the predecessor of $///$.
- ▶ **Definition 1.9** The following set of axioms are called the **Peano axioms** (Giuseppe Peano *1858, †1932)
- ▶ **Axiom 1.10 (P1)** “ ” (aka. “**zero**”) is a unary natural number. “ ” (aka. “zero”) is a unary natural number.
- ▶ **Axiom 1.11 (P2)** Every unary natural number has a successor that is a unary natural number and that is different from it.
- ▶ **Axiom 1.12 (P3)** Zero is not a successor of any unary natural number.
- ▶ **Axiom 1.13 (P4)** Different unary natural numbers have different successors.
- ▶ **Axiom 1.14 (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?)

3.2 Reasoning about Natural Numbers

Reasoning about Natural Numbers

- ▶ The Peano axioms can be used to reason about natural numbers.
- ▶ **Definition 2.1** An **axiom** (or **postulate**) is a statement about mathematical objects that we **assume to be true**.
- ▶ **Definition 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 2.3** We call a sequence of **inferences** a **derivation** or a **proof** (of the last statement).

Let's practice derivations and proofs

- ▶ **Example 2.4** `//////////` is a unary natural number
- ▶ **Theorem 2.5** `///` is a different unary natural number than `//`.
- ▶ **Theorem 2.6** `/////` is a different unary natural number than `///`.
- ▶ **Theorem 2.7** There is a unary natural number of which `///` is the successor
- ▶ **Theorem 2.8** There are at least 7 unary natural numbers.
- ▶ **Theorem 2.9** 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 2.10** 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 an 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 □

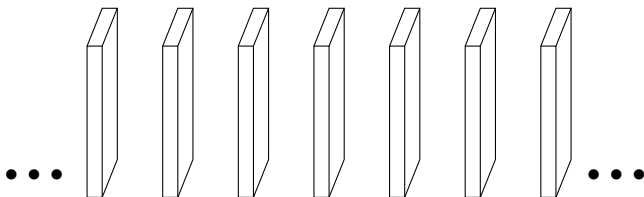
This seems awfully clumsy, lets introduce some notation

- ▶ **Idea:** we allow ourselves to give names to unary natural numbers (we use $n, m, l, k, n_1, n_2, \dots$ 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 2.11** 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 2.12** We denote the set of all unary natural numbers with \mathbb{N}_1 . (either representation)

The Domino Theorem

- ▶ **Theorem 2.13** 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.



The Domino Induction

- ▶ **Proof:** We prove the assertion by induction over i with the property P that “ S_i falls in the direction of $S_{s(i)}$ ”.
 - P.1 We have to consider two cases
 - P.1.1 **base case:** i is zero:
 - P.1.1.1 We have assumed that “ S_0 is pushed towards S_1 , so that it falls” □
 - P.1.2 **step case:** $i = s(j)$ for some unary natural number j :
 - P.1.2.1 We assume that P holds for S_j , i.e. S_j falls in the direction of $S_{s(j)} = S_i$.
 - P.1.2.2 But we know that S_j has the same weight as S_i , which is unstable,
 - P.1.2.3 so S_i falls into the direction opposite to S_j , i.e. towards $S_{s(i)}$ (we have a linear sequence of dominos) □
 - P.2 We have considered all the cases, so we have proven that P holds for all unary natural numbers i . (by induction)
 - P.3 Now, the assertion follows trivially, since if “ S_i falls in the direction of $S_{s(i)}$ ”, then in particular “ S_i falls”. □

3.3 Defining Operations on Natural Numbers

What can we do with unary natural numbers?

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

Addition on unary natural numbers is associative

- ▶ **Theorem 3.2** 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 $s((n \oplus m) \oplus l) = (n \oplus m) \oplus s(l)$ □



More Operations on Unary Natural Numbers

- ▶ **Definition 3.3** 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 3.4** 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 3.5** 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 3.6** The **unary product** operation can be defined by the equations $\bigodot_{i=o}^o n_i = s(o)$ and $\bigodot_{i=o}^{s(m)} n_i = n_{s(m)} \odot \bigodot_{i=o}^m n_i$.

3.4 Talking (and writing) about Mathematics

Talking about Mathematics (MathTalk)

- ▶ **Definition 4.1** Mathematicians use a stylized language that
 - ▶ uses formulae to represent mathematical objects, e.g. $\int_0^1 x^{3/2} dx$
 - ▶ uses **math idioms** for special situations (e.g. *iff, hence, let...be..., then...*)
 - ▶ classifies statements by role (e.g. **Definition, Lemma, Theorem, Proof, Example**)

We call this language **mathematical vernacular**.

- ▶ **Definition 4.2** Abbreviations for Mathematical statements in **MathTalk**
 - ▶ \wedge and “ \vee ” are common notations for “and” and “or”
 - ▶ “not” is in mathematical statements often denoted with \neg
 - ▶ $\forall x.P$ ($\forall x \in S.P$) stands for “condition P holds for all x (in S)”
 - ▶ $\exists x.P$ ($\exists x \in S.P$) stands for “there exists an x (in S) such that proposition P holds”
 - ▶ $\nexists x.P$ ($\nexists x \in S.P$) stands for “there exists no x (in S) such that proposition P holds”
 - ▶ $\exists^1 x.P$ ($\exists^1 x \in S.P$) stands for “there exists one and only one x (in S) such that proposition P holds”
 - ▶ “iff” as abbreviation for “if and only if”, symbolized by “ \Leftrightarrow ”
 - ▶ the symbol “ \Rightarrow ” is used as a shortcut for “implies”

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

- ▶ **Example 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

Peano Axioms in Mathtalk

- **Example 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
- $0 \in \mathbb{N}_1$ (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)
 - $\neg(\exists n \in \mathbb{N}_1. 0 = s(n))$ (zero is not a successor)
 - $\forall n \in \mathbb{N}_1. \forall m \in \mathbb{N}_1. n \neq m \Rightarrow s(n) \neq s(m)$ (different successors)
 - $\forall P. (P(0) \wedge (\forall n \in \mathbb{N}_1. P(n) \Rightarrow P(s(n)))) \Rightarrow (\forall m \in \mathbb{N}_1. P(m))$ (induction)

3.5 Naive Set Theory

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 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)

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**: $(A \supset B) :\Leftrightarrow (A \supseteq B) \wedge (A \neq B)$

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:** $A \times B := \{(a, b) \mid a \in A \wedge b \in B\}$, call (a, b) **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 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 $\bigcap_{i \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\}} S_i$.

Sizes of Sets

- ▶ We would like to talk about the size of a set. Let us try a definition
- ▶ **Definition 5.3** The **size** $\#(A)$ of a set A is the number of elements in A .
- ▶ **Conjecture 5.4** *Intuitively we should have the following identities:*
 - ▶ $\#\{a, b, c\} = 3$
 - ▶ $\#\mathbb{N} = \infty$ (*infinity*)
 - ▶ $\#(A \cup B) \leq \#(A) + \#(B)$ (\triangleleft *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*)

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

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)

3.6 Relations

- ▶ **Definition 6.1** $R \subseteq A \times B$ is a (binary) **relation** between A and B .
- ▶ **Definition 6.2** If $A = B$ then R is called a **relation on A** .
- ▶ **Definition 6.3** $R \subseteq A \times B$ is called **total** iff $\forall x \in A. \exists y \in B. (x, y) \in R$.
- ▶ **Definition 6.4** $R^{-1} := \{(y, x) \mid (x, y) \in R\}$ 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 $S \circ R := \{(a, c) \in A \times C \mid \exists b \in B. (a, b) \in R \wedge (b, c) \in S\}$
- ▶ **Example 6.5** relation $\subseteq, =, \textit{has_color}$
- ▶ **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 $(a, (b, c))$
 - ▶ we can (and often will) see $\langle a, b, c \rangle$ as $(a, (b, c))$ different representations of the same object.

Properties of binary Relations


- ▶ **Definition 6.6 (Relation Properties)** A relation $R \subseteq A \times A$ is called
 - ▶ **reflexive** on A , iff $\forall a \in A. (a, a) \in R$
 - ▶ **irreflexive** on A , iff $\forall a \in A. (a, a) \notin R$
 - ▶ **symmetric** on A , iff $\forall a, b \in A. (a, b) \in R \Rightarrow (b, a) \in R$
 - ▶ **asymmetric** on A , iff $\forall a, b \in A. (a, b) \in R \Rightarrow (b, a) \notin R$
 - ▶ **antisymmetric** on A , iff $\forall a, b \in A. ((a, b) \in R \wedge (b, a) \in R) \Rightarrow a = b$
 - ▶ **transitive** on A , iff $\forall a, b, c \in A. ((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$
 - ▶ **equivalence relation** on A , iff R is reflexive, symmetric, and transitive.
- ▶ **Example 6.7** The equality relation is an equivalence relation on any set.
- ▶ **Example 6.8** On sets of persons, the “mother-of” relation is an non-symmetric, non-reflexive relation.

Strict and Non-Strict Partial Orders

- ▶ **Definition 6.9** A relation $R \subseteq A \times A$ is called
 - ▶ **partial order** on A , iff R is reflexive, antisymmetric, and transitive on A .
 - ▶ **strict partial order** on A , iff it is irreflexive and transitive on A .
- ▶ In contexts, where we have to distinguish between strict and non-strict ordering relations, we often add an adjective like *non-strict* or *weak* or *reflexive* to the term *partial order*. We will usually write strict partial orderings with asymmetric symbols like \prec , and non-strict ones by adding a line that reminds of equality, e.g. \preceq .
- ▶ **Definition 6.10 (Linear order)** A partial order is called **linear** on A , iff all elements in A are **comparable**, i.e. if $(x, y) \in R$ or $(y, x) \in R$ for all $x, y \in A$.
- ▶ **Example 6.11** The \leq relation is a linear order on \mathbb{N} (all elements are comparable)
- ▶ **Example 6.12** The “ancestor-of” relation is a partial order that is not linear.
- ▶ **Lemma 6.13** *Strict partial orderings are asymmetric.*
- ▶ **Proof Sketch:** By contradiction: If $(a, b) \in R$ and $(b, a) \in R$, then $(a, a) \in R$ by transitivity □
- ▶ **Lemma 6.14** *If \preceq is a (non-strict) partial order, then $\prec := \{(a, b) \mid (a \preceq b) \wedge a \neq b\}$ is a strict partial order. Conversely, if \prec is a strict partial order, then $\preceq := \{(a, b) \mid (a \prec b) \vee a = b\}$ is a non-strict partial order.*

3.7 Functions

Functions (as special relations)

- ▶ **Definition 7.1** $f \subseteq X \times Y$, is called a **partial function**, iff for all $x \in X$ there is at most one $y \in Y$ with $(x, y) \in f$.
 - ▶ **Notation 7.2** $f: X \rightarrow Y; x \mapsto y$ if $(x, y) \in f$ (arrow notation)
 - ▶ call X the **domain** (write $\text{dom}(f)$), and Y the **codomain** ($\text{codom}(f)$) (come with f)
 - ▶ **Notation 7.3** $f(x) = y$ instead of $(x, y) \in f$ (function application)
- ▶ **Definition 7.4** We call a partial function $f: X \rightarrow Y$ **undefined at** $x \in X$, iff $(x, y) \notin f$ for all $y \in Y$. (write $f(x) = \perp$)
- ▶ **Definition 7.5** If $f: X \rightarrow Y$ is a total relation, we call f a **total function** and write $f: X \rightarrow Y$. ($\forall x \in X. \exists^1 y \in Y. (x, y) \in f$)
 - ▶ **Notation 7.6** $f: x \mapsto y$ if $(x, y) \in f$ (arrow notation)
- ▶ **Definition 7.7** The **identity function** on a set A is defined as $\text{Id}_A := \{(a, a) \mid a \in A\}$.
 **Warning:** 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 7.8** Given sets A and B We will call the set $A \rightarrow B$ ($A \rightharpoonup B$) of all (partial) functions from A to B the (partial) **function space** from A to B .
- ▶ **Example 7.9** Let $\mathbb{B} := \{0, 1\}$ be a two-element set, then

$$\mathbb{B} \rightarrow \mathbb{B} = \{\{(0, 0), (1, 0)\}, \{(0, 1), (1, 1)\}, \{(0, 1), (1, 0)\}, \{(0, 0), (1, 1)\}\}$$

$$\mathbb{B} \rightharpoonup \mathbb{B} = \mathbb{B} \rightarrow \mathbb{B} \cup \{\emptyset, \{(0, 0)\}, \{(0, 1)\}, \{(1, 0)\}, \{(1, 1)\}\}$$

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

Lambda-Notation for Functions I

- ▶ **Problem:** In maths we write $f(x) := x^2 + 3x + 5$ to define a function f , then we can talk about $\mathbf{dom}(f)$. But if we do not want to use a name, we can only say $\mathbf{dom}(\{(x, y) \in \mathbb{R} \times \mathbb{R} \mid y = x^2 + 3x + 5\})$
- ▶ **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 7.10** To make the role of arguments extremely clear, we write functions in **λ -notation**. For $f = \{(x, E) \mid x \in X\}$, where E is an expression, we write $\lambda x \in X. E$.

Lambda-Notation for Functions II

- ▶ **Example 7.11** The simplest function we always try everything on is the identity function:

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

- ▶ **Example 7.12** We can also to more complex expressions, here we take the square function

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

- ▶ **Example 7.13** λ -notation also works for more complicated domains. In this case we have pairs as arguments.

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

Properties of functions, and their converses

- ▶ **Definition 7.14** 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.
- ▶ **Observation 7.15** If f is injective, then the converse relation f^{-1} is a partial function.
- ▶ **Observation 7.16** If f is surjective, then the converse f^{-1} is a total relation.
- ▶ **Definition 7.17** If f is bijective, call the converse relation **inverse function**, we (also) write it as f^{-1} .
- ▶ **Observation 7.18** If f is bijective, then f^{-1} is a total function.
- ▶ **Observation 7.19** If $f: A \rightarrow B$ is bijective, then $f \circ f^{-1} = Id_A$ and $f^{-1} \circ f = Id_B$.
- ▶ **Example 7.20** 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 you know from elementary school.

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 7.21** 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 7.22** 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 7.23** We have the following identities for finite sets A and B
 - ▶ $\#\{a, b, c\} = 3$ (e.g. choose $f = \{(a, 0), (b, 1), (c, 2)\}$)
 - ▶ $\#(A \cup B) \leq \#(A) + \#(B)$
 - ▶ $\#(A \cap B) \leq \min(\#(A), \#(B))$
 - ▶ $\#(A \times B) = \#(A) \cdot \#(B)$
- ▶ With the definition above, we can prove them (last three \leadsto Homework)

Operations on Functions

- ▶ **Definition 7.24** 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).

- ▶ **Definition 7.25** Let $f \in A \rightarrow B$ and $C \subseteq A$, then we call the function $f|_C := \{(c, b) \in f \mid c \in C\}$ the **restriction** of f to C .
- ▶ **Definition 7.26** Let $f: A \rightarrow B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call
 - ▶ $f(A') := \{b \in B \mid \exists a \in A'. (a, b) \in f\}$ the **image** of A' under f ,
 - ▶ $\text{Im}(f) := f(A)$ the **image** of f , and
 - ▶ $f^{-1}(B') := \{a \in A \mid \exists b \in B'. (a, b) \in f\}$ the **pre-image** of B' under f

Chapter 4 Computing with Functions over Inductively Defined Sets

4.1 Standard ML: A Functional Programming Language

Enough theory, let us start computing with functions

- ▶ We will use Standard ML in this course.
- ▶ We call programming languages where procedures can be fully described in terms of their input/output behavior **functional**.
- ▶ **But most importantly...**: ...it emphasizes “thinking” over “hacking”.

Standard ML (SML)

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

Programming in SML (Basic Language)

- ▶ **Generally:** start the SML interpreter, play with the program state.
- ▶ **Definition 1.1 (Predefined objects in SML)** (SML comes with a basic inventory)
 - ▶ **basic types** int, real, bool, string, ...
 - ▶ **basic type constructors** \rightarrow , *
 - ▶ **basic operators** numbers, true, false, +, *, -, >, ^, ... (⚠ overloading)
 - ▶ **control structures** **if** ϕ **then** E_1 **else** E_2 ;
 - ▶ **comments** (*this is a comment *)

Programming in SML (Declarations)

- ▶ **Definition 1.2** **declarations** bind **variables** (abbreviations for convenience)
 - ▶ **value declarations** e.g. **val** pi = 3.1415;
 - ▶ **type declarations** e.g. **type** twovec = int * int;
 - ▶ **function declarations** e.g. **fun** square (x:real) = x*x; (leave out type, if unambiguous)

A function declaration only declares the **function name** as a globally visible name. The **formal parameters** in brackets are only visible in the **function body**.

- ▶ 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.
- ▶ **Definition 1.3** A **local declaration** uses **let** to bind variables in its **scope** (delineated by **in** and **end**).
- ▶ **Example 1.4** Local definitions can shadow existing variables.

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

Programming in SML (Component Selection)

- ▶ **Definition 1.5** Using structured patterns, we can declare more than one variable. We call this **pattern matching**.

- ▶ **Example 1.6 (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 1.7 anonymous variables** (if we are not interested in one value)

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

- ▶ **Example 1.8** We can define the selector function for pairs in SML as

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

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

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

What's next?

More SML constructs and general theory of functional programming.

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?)

Defining Functions by Recursion

- ▶ **Observation:** 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.
- ▶ **Example 1.9** We have the following evaluation trace with result [2,3,4]

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

- ▶ **Definition 1.10** 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

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

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 1.11** Flattening lists of lists (using the infix append operator `@`)

```
fun flat [] = [] (* base case *)  
  | flat (h::t) = h @ flat t; (* step case *)  
val flat = fn : 'a list list -> 'a list
```

Let's test it on an argument:

```
flat [ ["When", "shall"], ["we", "three"], ["meet", "again"] ];  
["When", "shall", "we", "three", "meet", "again"]
```

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)**

Higher-Order Functions

▶ **Idea:** pass functions as arguments (functions are normal values.)

▶ **Example 1.12** Mapping a function over a list

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

▶ **Example 1.13** We can program the map function ourselves!

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

▶ **Example 1.14** declaring functions (yes, functions are normal values.)

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

▶ **Example 1.15** returning functions: (again, functions are normal values.)

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


Cartesian and Cascaded Functions

- ▶ We have not been able to treat binary, ternary, . . . functions directly
- ▶ **Workaround 1:** Make use of (Cartesian) products (unary functions on tuples)
- ▶ **Example 1.16** $+: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ with $+\left(\left(3, 2\right)\right)$ 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 1.17** $+: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ with $+(3)(2)$ instead of $+\left(\left(3, 2\right)\right)$.

```
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.

Cartesian and Cascaded Functions (Brackets)

- ▶ **Definition 1.18** Call a function **Cartesian**, iff the argument type is a product type, call it **cascaded**, iff the result type is a function type.

- ▶ **Example 1.19** 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

- ▶ ▶ $e_1 e_2 e_3 \rightsquigarrow (e_1 e_2) e_3$ (function 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;
```

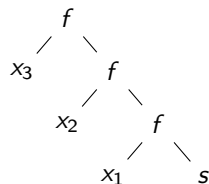
- ▶ Another simplification (related to those above)

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

Folding Operators

- ▶ **Definition 1.20** SML provides the **left folding operator** to realize a recurrent computation schema

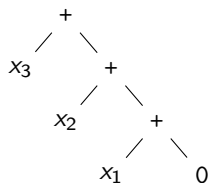
$\text{foldl} : ('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b$
 $\text{foldl } f \ s \ [x_1, x_2, x_3] = f(x_3, f(x_2, f(x_1, s)))$



We call the function f the **iterator** and s the **start value**

- ▶ **Example 1.21** Folding the iterator **op+** with start value 0:

$\text{foldl } \text{op+ } 0 \ [x_1, x_2, x_3] = x_3 + (x_2 + (x_1 + 0))$

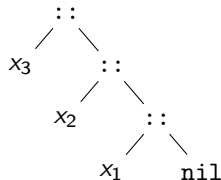


Thus the function given by the expression $\text{foldl } \text{op+ } 0$ adds the elements of integer lists.

Folding Procedures (continued)

► Example 1.22 (Reversing Lists)

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

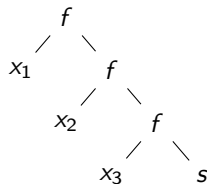


Thus the procedure `fun rev xs = foldl op:: nil xs` reverses a list

Folding Procedures (foldr)

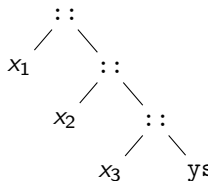
- ▶ **Definition 1.23** The **right folding operator** foldr is a variant of foldl that processes the list elements in reverse order.

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



- ▶ **Example 1.24 (Appending Lists)**

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



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

Now that we know some SML

SML is a “functional Programming Language”

What does this all have to do with functions?

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

4.2 Inductively Defined Sets and Computation

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
- ▶ $+(n, 0) = n$ (base) and $+(m, s(n)) = s(+((m, n)))$ (step)
- ▶ **Theorem 2.1** $+\subseteq (\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1$ is a total function.
- ▶ We have to show that for all $(n, m) \in \mathbb{N}_1 \times \mathbb{N}_1$ there is exactly one $l \in \mathbb{N}_1$ with $((n, m), l) \in +$.
- ▶ We will use functional notation for simplicity

Addition is a total Function

► **Lemma 2.2** For all $(n, m) \in \mathbb{N}_1 \times \mathbb{N}_1$ there is exactly one $l \in \mathbb{N}_1$ with $+((n, m)) = l$.

► **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 $+((n, o)) = n = l$.

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

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

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

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

□

► **Corollary 2.3** $+: \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$ is a total function.

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 $+$: $+((n, o)) = n$ (base) and $+((m, s(n))) = s(+((m, n)))$ (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 2.4** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$
- ▶ **Example 2.5** 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

Inductively Defined Sets

- ▶ **Definition 2.6** An **inductively defined set** $\langle S, C \rangle$ is a set S together with a finite set $C := \{c_i \mid 1 \leq i \leq n\}$ of k_i -ary **constructors** $c_i: S^{k_i} \rightarrow S$ 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)
 - ▶ $\text{Im}(c_i) \cap \text{Im}(c_j) = \emptyset$ for $i \neq j$, and (no confusion between constructors)
 - ▶ for every $s \in S$ there is a constructor $c \in C$ with $s \in \text{Im}(c)$. (no junk)
- ▶ Note that we also allow nullary constructors here.
- ▶ **Example 2.7** $\langle \mathbb{N}_1, \{s, o\} \rangle$ is an inductively defined set.
- ▶ **Proof:** We check the three conditions in Definition 2.6 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**. □

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 2.8** e.g. $[3, 2, 1] \hat{=} \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{nil})))$ and $[] \hat{=} \text{nil}$
- ▶ **Definition 2.9** 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 1.9.
- ▶ **Axiom 2.10 (LP1)** $\text{nil} \in \mathcal{L}[\mathbb{N}]$ (generation axiom (nil))
- ▶ **Axiom 2.11 (LP2)** $\text{cons}: \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \rightarrow \mathcal{L}[\mathbb{N}]$ (generation axiom (cons))
- ▶ **Axiom 2.12 (LP3)** nil is not a cons-value
- ▶ **Axiom 2.13 (LP4)** cons is injective
- ▶ **Axiom 2.14 (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 Pthen every list $l \in \mathcal{L}[\mathbb{N}]$ has property P .

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: $\text{nil} @ l = l$ and $\text{cons}(n, l) @ r = \text{cons}(n, l @ r)$
- ▶ **Example 2.15** $[3, 2, 1] @ [1, 2] = [3, 2, 1, 1, 2]$ and
 $[] @ [1, 2, 3] = [1, 2, 3] = [1, 2, 3] @ []$
- ▶ **Lemma 2.16** 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: $l = \text{nil}$: must have $s = r$.
 - P.1.2 step case: $l = \text{cons}(n, k)$ for some list k :
 - P.1.2.1 Assume that here is a unique s' with $s' = k @ r$,
 - P.1.2.2 then $s = \text{cons}(n, k) @ r = \text{cons}(n, k @ r) = \text{cons}(n, s')$.
- ▶ **Corollary 2.17** Append is a function (see, this just worked fine!)

Operations on Lists: more examples

- ▶ **Definition 2.18** $\lambda(\text{nil}) = o$ and $\lambda(\text{cons}(n, l)) = s(\lambda(l))$
- ▶ **Definition 2.19** $\rho(\text{nil}) = \text{nil}$ and $\rho(\text{cons}(n, l)) = \rho(l) @ \text{cons}(n, \text{nil})$.

4.3 Inductively Defined Sets in SML

Data Type Declarations I

- ▶ **Definition 3.1** SML **data type** provide concrete syntax for inductively defined sets via the keyword **datatype** followed by a list of **constructor declarations**.
- ▶ **Example 3.2** We can declare a data type for unary natural numbers by
 - **datatype** mynat = zero | suc **of** mynat;
 - datatype** mynat = suc **of** mynat | zerothis gives us constructor functions $\text{zero} : \text{mynat}$ and $\text{suc} : \text{mynat} \rightarrow \text{mynat}$.
- ▶ **Observation 3.3** *We can define functions by (complete) case analysis over the constructors*
- ▶ **Example 3.4 (Converting types)**
 - fun** num (zero) = 0 | num (suc(n)) = num(n) + 1;
 - val** num = **fn** : mynat \rightarrow int

Data Type Declarations II

▶ Example 3.5 (Missing Constructor Cases)

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

▶ Example 3.6 (Inconsistency)

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

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

Data Types Example (Geometric Shapes)

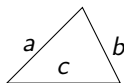
- describe three kinds of geometrical forms as mathematical objects



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 \rightarrow shape

Square : real \rightarrow shape

Triangle : real * real * real \rightarrow shape

Data Types Example (Areas of Shapes)

- ▶ a procedure that computes the area of a shape:

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

New Construct: Standard structure Math

(see [SML10])

- ▶ some experiments

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

Chapter 5 A Theory of SML: Abstract Data Types and Term Languages

What's next?

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

5.1 Abstract Data Types and Ground Constructor Terms

Abstract Data Types (ADT)

- ▶ **Definition 1.1** 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 sorts are sorts)
 - ▶ If $\mathbb{A}, \mathbb{B} \in \mathcal{S}$, then $\mathbb{A} \times \mathbb{B} \in \mathcal{S}$ (product sorts are sorts)
 - ▶ If $\mathbb{A} \in \mathcal{S}$ and $\mathbb{B} \in \mathcal{S}^0$, then $\mathbb{A} \rightarrow \mathbb{B} \in \mathcal{S}$ (function sorts are sorts)
- ▶ **Definition 1.2** 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 1.3** 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 1.4** $\mathcal{B} := \langle \{\mathbb{B}\}, \{[T: \mathbb{B}], [F: \mathbb{B}]\} \rangle$ is an abstract data for **truth values**.
- ▶ **Example 1.5** $\langle \{\mathbb{N}\}, \{[o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}]\} \rangle$ represents unary natural numbers.
- ▶ **Example 1.6**
 $\mathcal{L} := \langle \{\mathbb{N}, \mathcal{L}(\mathbb{N})\}, \{[o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}], [\text{nil}: \mathcal{L}(\mathbb{N})], [\text{cons}: \mathbb{N} \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})]\} \rangle$ In particular, the term $\text{cons}(s(o), \text{cons}(o, \text{nil}))$ represents the list $[1, 0]$
- ▶ **Example 1.7** $\langle \{\mathcal{S}^0, \mathcal{S}\}, \{[\iota: \mathcal{S}^0 \rightarrow \mathcal{S}], [\rightarrow: \mathcal{S} \times \mathcal{S}^0 \rightarrow \mathcal{S}], [\times: \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}]\} \rangle$
(what is this?)

Ground Constructor Terms

- ▶ **Definition 1.8** 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 $\mathcal{T}^g(\mathcal{A}) := \bigcup_{\mathbb{A} \in \mathcal{S}} \mathcal{T}_{\mathbb{A}}^g(\mathcal{A})$.

- ▶ **Example 1.9** $\text{cons}(s(o), \text{nil}) \in \mathcal{T}_{\mathcal{L}(\mathbb{N})}^g(\mathcal{L})$ where \mathcal{L} is the ADT from Example 1.6.
- ▶ **Definition 1.10** If $t = c(t')$ then we say that the symbol c is the **head** of t (write $\text{head}(t)$). If $t = a$, then $\text{head}(t) = a$; $\text{head}(\langle t_1, t_2 \rangle)$ is undefined.
- ▶ **Notation 1.11** We will write $c(a, b)$ instead of $c(\langle a, b \rangle)$ (cf. **binary function**)

Peano Axioms for Sorts in Abstract Data Types

- ▶ **Observation 1.12** *The set $\mathcal{T}^g(\mathcal{A})$ of ground constructor terms over an abstract data type \mathcal{A} form an inductively defined set.*
- ▶ **Proof Sketch:** We just think of each of the clauses in the definition as constructors: “constants”, “pairs”, and “constructor applications”. □

- ▶ **Example 1.13** We can even program it in SML with the datatype constructor:

```
datatype basesort = a | b | c | ...
```

```
datatype sort = base of basesort | pairsort of sort * sort | funsort of sort * sort
```

```
symbol = string
```

```
datatype gct = const of symbol | pair of gct * gct | app of symbol * gct
```

With this, the term $\text{cons}(s(o), \text{nil}) \in \mathcal{T}_{\mathcal{L}(\mathbb{N})}^g(\mathfrak{L})$ from Example 1.9 is represented as `app("cons",app("suc",const("zero")),const"nil")`.

- ▶ **Idea:** In abstract data types we have a kind of Peano Axioms as well.
- ▶ **Axiom 1.14** if t is a ground constructor term of sort \mathbb{T} , then $t \in \mathbb{T}$
- ▶ **Axiom 1.15** equality on ground constructor terms is trivial
- ▶ **Axiom 1.16** only ground constructor terms of sort \mathbb{T} are in \mathbb{T} (induction axioms)

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 math talk: \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 1.17** 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.

5.2 A First Abstract Interpreter

But how do we compute?

- ▶ **Problem:** We can **define** functions, but how do we compute them?
- ▶ **Intuition:** We direct the equations (l2r) and use them as rules.
- ▶ **Definition 2.1** 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 2.2** 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 2.3** Let $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D} \rangle$, 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 2.4** 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 2.5** An **abstract computation** is a computation that we can perform in our heads. (**no real world constraints like memory size, time limits**)
- ▶ **Definition 2.6** An **abstract interpreter** is an imagined machine that performs (abstract) computations, given abstract procedures.

Example: the functions ρ and $@$ on lists

- ▶ **Example 2.7** Consider the abstract procedures
 $\langle \rho :: \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}) ; \{ \rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil})), \rho(\text{nil}) \rightsquigarrow \text{nil} \} \rangle$
 $\langle @ :: \mathcal{L}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}) ; \{ @(\text{cons}(n, l), r) \rightsquigarrow \text{cons}(n, @(l, r)), @(nil, l) \rightsquigarrow l \} \rangle$
- ▶ 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 $\text{cons}(2, \text{cons}(1, \text{nil}))$

An Abstract Interpreter (preliminary version)

- ▶ **Definition 2.8 (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** $a \in \mathcal{T}_{\mathbb{A}}^{\mathbb{A}}(\mathcal{A})$.
 - ▶ **Output:** a **result** $r \in \mathcal{T}_{\mathbb{R}}^{\mathbb{R}}(\mathcal{A})$.
 - ▶ **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 2.9** We say that an abstract procedure $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ **terminates** (on $a \in \mathcal{T}_{\mathbb{A}}^{\mathbb{A}}(\mathcal{A})$), iff the computation (starting with $f(a)$) reaches a state, where no rule applies.

An Abstract Interpreter (preliminary version)

- ▶ **Definition 2.10 (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** $a \in \mathcal{T}_{\mathbb{A}}^{\mathbb{A}}(\mathcal{A})$.
 - ▶ **Output:** a **result** $r \in \mathcal{T}_{\mathbb{R}}^{\mathbb{R}}(\mathcal{A})$.
 - ▶ **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 2.11** We say that an abstract procedure $\langle f::\mathbb{A} \rightarrow \mathbb{R}; \mathcal{R} \rangle$ **terminates** (on $a \in \mathcal{T}_{\mathbb{A}}^{\mathbb{A}}(\mathcal{A})$), iff the computation (starting with $f(a)$) reaches a state, where no rule applies.
 - ▶ There are a lot of words here that we **do not understand**
 - ▶ let us try to understand them better \rightsquigarrow more theory!

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 2.12** 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.
- ▶ **Example 2.13** $x_{\mathbb{A}}, n_{\mathbb{N}}, x_{\mathbb{C}^3}, \dots$ are variable .s
- ▶ **Definition 2.14** 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 2.15** If t is a constructor term, then we denote the set of variables occurring in t with $\mathbf{free}(t)$. If $\mathbf{free}(t) = \emptyset$, then we say t is **ground** or **closed**.

Constr. Terms with Variables: The Complete Definition

- **Definition 2.16** 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 2.17** We define the **depth** of a constructor term $t \in \mathcal{T}(\mathcal{A}; \mathcal{V})$ by the way it is constructed.

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

- **Observation 2.18** *This is a recursive function on the inductively defined set $\mathcal{T}(\mathcal{A}; \mathcal{V})$. (made possible by Peano axioms)*

5.3 Substitutions

- ▶ **Definition 3.1** 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 $\mathbf{supp}(\sigma) := \{x_{\mathbb{A}} \in \mathcal{V}_{\mathbb{A}} \mid \sigma(x_{\mathbb{A}}) \neq x_{\mathbb{A}}\}$ is finite and $\sigma(x_{\mathbb{A}}) \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$. $\mathbf{supp}(\sigma)$ is called the **support** of σ .
- ▶ **Definition 3.2** If $\mathbf{supp}(\sigma) = \emptyset$, then we call σ the **empty substitution** and write σ as ϵ .
- ▶ **Notation 3.3** We denote the substitution σ with $\mathbf{supp}(\sigma) = \{x_{\mathbb{A}_i}^i \mid 1 \leq i \leq n\}$ 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 3.4 (Substitution Extension)** Let σ be a substitution, then we denote with $\sigma, [t/x_{\mathbb{A}}]$ the function $\{\langle y_{\mathbb{B}}, t \rangle \in \sigma \mid y_{\mathbb{B}} \neq x_{\mathbb{A}}\} \cup \{\langle x_{\mathbb{A}}, t \rangle\}$. ($\sigma, [t/x_{\mathbb{A}}]$ coincides with σ off $x_{\mathbb{A}}$, and gives the result t there.)
- ▶ **Note:** If σ is a substitution, then $\sigma, [t/x_{\mathbb{A}}]$ is also a substitution.

Substitution Application

- ▶ **Definition 3.5 (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 3.6** 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 3.7** $[a/x], [f(b)/y], [a/z]$ instantiates $g(x, y, h(z))$ to $g(a, f(b), h(a))$.
(sorts elided here)
- ▶ **Definition 3.8** We give the **defining equations for substitution application** on an abstract data type $\mathcal{A} := \langle \mathcal{S}^0, \mathcal{D} \rangle$:
 - ▶ $\sigma(c) = c$ if $[c: \mathbb{T}] \in \mathcal{D}$.
 - ▶ $\sigma(x_{\mathbb{A}}) = t$ 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.

Substitution Application conserves Sorts

- ▶ **Theorem 3.9** Let \mathcal{A} be an abstract data type, $t \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$, and σ a substitution on \mathcal{A} , then $\sigma(t) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$.
- ▶ **Proof:** by induction on $\text{dp}(t)$ using Definition 2.16 and Definition 3.5
 - P.1 By Definition 2.16 we have to consider four cases
 - P.1.1 $[t: \mathbb{T}] \in \mathcal{D}$: $\sigma(t) = t$ by Definition 3.5, so $\sigma(t) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$ by construction.
 - P.1.2 $t \in \mathcal{V}_{\mathbb{T}}$: We have $\sigma(t) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$ by Definition 3.1,
 - 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 $(c(\sigma(a))) \in \mathcal{T}_{\mathbb{T}}(\mathcal{A}; \mathcal{V})$. □

Uniqueness of Matchers

- ▶ **Theorem 3.10** For any $s, t \in \mathcal{T}(\mathcal{A}; \mathcal{V})$, there is at most one σ with $\sigma(s) = t$.
- ▶ **Proof:** We prove this by induction on s (using the Peano Axioms for $\mathcal{T}(\mathcal{A}; \mathcal{V})$)
 - P.1 We have four cases to consider
 - P.1.1 s is a constant $([c: \mathbb{T}] \in \mathcal{D})$: If $t = c$, then $\sigma = \epsilon$, else no matcher exists. □
 - P.1.2 s is a variable $x_{\mathbb{T}}$: Here σ must be $[t/x_{\mathbb{T}}]$. □
 - P.1.3 s is a pair $\langle a, b \rangle$:
 - P.1.3.1 Then t must be of the form $\langle c, d \rangle$ for some terms c and d by Definition 3.8.
 - P.1.3.2 By inductive hypothesis, we have at most one matcher σ_a and σ_b with $\sigma_a(a) = c$ and $\sigma_b(b) = d$ respectively.
 - P.1.3.3 Now let $C := \mathbf{supp}(\sigma_a, \sigma_b)$. If $\sigma_a|_C = \sigma_b|_C$, then $\sigma := \sigma_a \cup \sigma_b$ instantiates $s = \langle a, b \rangle$ to $t = \langle c, d \rangle$, otherwise no matcher exists. □
 - P.1.4 s is an application $f(a)$ with $[f: \mathbb{A} \rightarrow \mathbb{T}] \in \mathcal{D}$ and $a \in \mathcal{T}_{\mathbb{A}}(\mathcal{A}; \mathcal{V})$:
 - P.1.4.1 Then t must be of the form $f(b)$ for some term b by Definition 3.8.
 - P.1.4.2 By inductive hypothesis, we have at most one matcher ρ with $\rho(a) = b$.
 - P.1.4.3 σ must be equal to ρ if that exists, since $\sigma(s) = \sigma(f(a)) = f(\sigma(a)) = f(\rho(a)) = f(b) = t$. □

Note: that we used two different inductions in the proofs of these two theorems: For the proof of Theorem 3.9, we used an “induction over the depth of [a constructor term]”, which is really an induction on natural numbers and correspondingly uses the induction axiom for natural numbers. Here we use the trick of making the property of all constructor terms we want to prove a property of natural numbers by involving the depth function: The property P is that “substitution application conserves sorts on *all constructor terms of depth n* ”. The four cases in the recursive definition of dp in Definition 2.17 give rise to four cases in the proof: two for the base case $n = 0$ and two for the step case $n > 0$. For the proof of Theorem 3.10 we directly used the induction axiom for the inductively defined set $\mathcal{T}(\mathcal{A}; \mathcal{V})$. Here the induction is over the “constructors” of $\mathcal{T}(\mathcal{A}; \mathcal{V})$, which correspond to the four cases in Definition 2.16. So even though the two proofs start with very different induction axioms, they end up with a very similar case analysis. Some authors like the directness of “structural induction” (induction over the structure of terms), while some prefer the more “elementary” nature of natural numbers induction. We have used both here to

expose the two methods.

5.4 Terms in Abstract Data Types

Are Constructor Terms Really Enough for Rules?

- ▶ **Example 4.1** $\rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, \text{nil}))$. (ρ is not a constructor)
- ▶ **Idea:** need to include symbols for the defined procedures. (provide declarations)
- ▶ **Definition 4.2** 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 4.3** 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 $[f: \mathbb{A} \rightarrow \mathbb{T}] \in \Sigma$.

Terms: The Complete Definition

- ▶ **Idea:** treat procedures (from Σ) and constructors (from \mathcal{D}) at the same time.
- ▶ **Definition 4.4** 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 $\mathcal{T}(\mathcal{A}, \Sigma; \mathcal{V})$.

Subterms

- ▶ **Idea:** Well-formed parts of constructor terms are constructor terms again (**maybe of a different sort**)
- ▶ **Definition 4.5** Let \mathcal{A} be an abstract data type and s , t , 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 4.6** 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' .
- ▶ **Note:** that we see a recursive definition of a relation in Maths here – recursion is not restricted to computation or computer science.
- ▶ **Example 4.7** $f(a)$ is a subterm of the terms $f(a)$ and $h(g(f(a), f(b)))$, and an immediate subterm of $h(f(a))$.

5.5 A Second Abstract Interpreter

Abstract Procedures, Final Version

- ▶ **Definition 5.1 (Rules, final version)** Let $\mathcal{A} := \langle S^0, \mathcal{D} \rangle$ be an abstract data type, Σ a signature over \mathcal{A} , and $f \notin (\mathbf{dom}(\mathcal{D}) \cup \mathbf{dom}(\Sigma))$ a symbol, then we call $f(s) \rightsquigarrow r$ 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 5.2 (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 5.3** Let \mathcal{A} be the union of the abstract data types from Example 1.6 and Example 1.17, 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 $[\mu: \mathbb{N} \times \mathcal{L}(\mathbb{N}) \rightarrow \mathbb{B}]$

Abstract Programs

- ▶ **Definition 5.4 (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 5.5** The two abstract procedures from Example 2.7
 - $\langle @:: \mathcal{L}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{ @(\text{cons}(n, l), r) \rightsquigarrow \text{cons}(n, @(l, r)), @(nil, l) \rightsquigarrow l \} \}$
 - $\langle \rho:: \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N}); \{ \rho(\text{cons}(n, l)) \rightsquigarrow @(\rho(l), \text{cons}(n, nil)), \rho(nil) \rightsquigarrow nil \} \}$constitute an abstract program over the abstract data type from Example 1.6:
$$\langle \{ \mathbb{N}, \mathcal{L}(\mathbb{N}) \}, \{ [o: \mathbb{N}], [s: \mathbb{N} \rightarrow \mathbb{N}], [nil: \mathcal{L}(\mathbb{N})], [\text{cons}: \mathbb{N} \times \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})] \} \rangle$$

An Abstract Interpreter (second version)

- ▶ **Definition 5.6 (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 5.7** 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?

5.6 Evaluation Order and Termination

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.

- ▶ ▶ the rule is unique, if the left-hand sides are non-overlapping (SML enforces this)
- ▶ ▶ the matcher is unique by Theorem 3.10
- ▶ **Observation 6.1** *sometimes there we can choose more than one subterm s and rule, and the computation and even the result differ.*

▶ Example 6.2

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

Idea: Prescribe the “choice” of subterm

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

An abstract call-by-value Interpreter (final)

- ▶ **Definition 6.3 (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)

Analyzing Termination of Abstract Procedures

- ▶ **Example 6.4** $\tau: \mathbb{N}_1 \rightarrow \mathbb{N}_1$, 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 6.5** 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 6.6** $\langle 4, 3 \rangle$ 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 6.7** 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.

- ▶ **Definition 6.8** 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 6.9** We say that R **terminates (on $a \in \mathbb{A}$)**, iff there is no infinite chain in R (that begins with a).
- ▶ **Definition 6.10** \mathcal{P} **diverges (on $a \in \mathbb{A}$)**, iff it does not terminate on a .
- ▶ **Theorem 6.11** 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 6.12** 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 6.13** 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} .

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 6 More SML

6.1 Recursion in the Real World

Consider the Fibonacci numbers

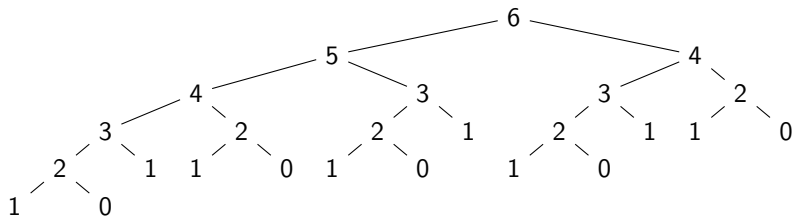
► **Fibonacci sequence:** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

► 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: $\{(n, n-1), (n, n-2) \mid n \in \mathbb{N}\}$ (it is a tree!)



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)

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;
```

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)
```

We trace the computation:

$\text{even}(4) \rightsquigarrow \text{odd}(3) \rightsquigarrow \text{even}(2) \rightsquigarrow \text{odd}(1) \rightsquigarrow \text{even}(0) \rightsquigarrow \text{true}$

6.2 Programming with Effects: Imperative Features in SML

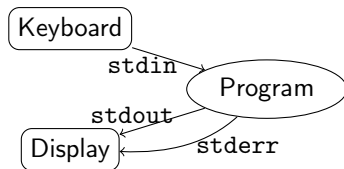
Programming with Effects on the Machine State

- ▶ 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!

6.2.1 Input and Output

Input and Output in SML I

- ▶ 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 2.1 (Input)

via `TextIO.inputLine : TextIO.instream -> string`

```
- TextIO.inputLine(TextIO.stdIn);  
  sdfkjsdlfkj
```

```
val it = "sdfkjsdlfkj" : string
```

▶ Example 2.2 (Printing to Standard Output)

`TextIO.print` prints its argument to `stdIn`

($\hat{=}$ screen)

```
print "sdfsfsdf"
```


Input and Output in SML II

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 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)

6.2.2 Programming with Exceptions

Raising Exceptions I

- ▶ **Idea:** Exceptions are generalized error codes
- ▶ **Definition 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 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.

Raising Exceptions II

- ▶ **Example 2.6** user-defined exceptions (exceptions are first-class objects)

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

- ▶ **Example 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
```

Programming with Exceptions

- ▶ **Example 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**

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
```


Handling Exceptions I

- ▶ **Definition 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 2.10 (SML Construct)** **exception handler:** exp **handle** rules
- ▶ **Example 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!"
```

Handling Exceptions II

- ▶ **Example 2.12** the `read_integer` function (just to be complete)

```
exception NaN; (* Not a Number *)
fun read_integer () =
  let
    val intstring = case TextIO.inputLine(TextIO.stdIn) of
                      NONE => raise NaN
                      | SOME(s) => s
  in
    case Int.fromString intstring of
      NONE => raise NaN
      | SOME i => i
  end
```

Using Exceptions for Optimizing Computation

- ▶ **Example 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
```

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

Part II Syntax and Semantics

Chapter 7 Encoding Programs as Strings

7.1 Formal Languages

The Mathematics of Strings

- ▶ **Definition 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 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 1.3** We will often write a string $\langle c_1, \dots, c_n \rangle$ as " $c_1 \dots c_n$ ", for instance "**abc**" for $\langle a, b, c \rangle$
- ▶ **Example 1.4** Take $A = \{h, 1, /\}$ as an alphabet. Each of the symbols h , 1 , and $/$ is a character. The vector $\langle /, /, 1, h, 1 \rangle$ is a string of length 5 over A .
- ▶ **Definition 1.5 (String Length)** Given a string s we denote its length with $|s|$.
- ▶ **Definition 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.
 $\text{conc}(\text{"text"}, \text{"book"}) = \text{"text"} + \text{"book"} = \text{"textbook"}$)

- ▶ **Definition 1.7** Let A be an alphabet, then we define the sets $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ of **nonempty strings** and $A^* := A^+ \cup \{\epsilon\}$ of **strings**.
- ▶ **Example 1.8** If $A = \{a, b, c\}$, then $A^* = \{\epsilon, a, b, c, aa, ab, ac, ba, \dots, aaa, \dots\}$.
- ▶ **Definition 1.9** A set $L \subseteq A^*$ is called a **formal language** in A .
- ▶ **Definition 1.10** We use $c^{[n]}$ for the string that consists of n times c .
- ▶ **Example 1.11** $\#^{[5]} = \langle \#, \#, \#, \#, \# \rangle$
- ▶ **Example 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 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 $\text{conc}(L_1, L_2) := \{s_1s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$.

Substrings and Prefixes of Strings

- ▶ **Definition 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 1.15** $\text{conc}(/, 1, h)$ is a substring of $\text{conc}(/, /, 1, h, 1)$, whereas $\text{conc}(/, 1, 1)$ is not.
- ▶ **Definition 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 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.

Lexical Order

- ▶ **Definition 1.18** Let A be an alphabet and \prec a strict partial order on A , Then we define a relation \prec_{lex} on A^* by

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

for $s, t \in A^*$. We call \prec_{lex} the **lexical order** induced by \prec on A^* .

- ▶ **Theorem 1.19** \prec_{lex} is a strict partial order on A^* . Moreover, if \prec is linear on A , then \prec_{lex} is linear on A^* .
- ▶ **Example 1.20** Roman alphabet with $a \prec b \prec c \cdots \prec z \rightsquigarrow$ telephone book order
(*computer* \prec_{lex} *text*, *text* \prec_{lex} *textbook*)

7.2 Elementary Codes

Character Codes

- ▶ **Definition 2.1** Let A and B be alphabets, then we call an injective function $c: A \rightarrow B^+$ a **character code**. A string $c(w) \in \{c(a) \mid a \in A\}$ is called a **codeword**.
- ▶ **Definition 2.2** A code is called **binary** iff $B = \{0, 1\}$.
- ▶ **Example 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 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 2.5** The extension c' of c from the above example on the string "bbabc"

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

- ▶ **Definition 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.,

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

Morse Code


- ▶ In the early days of telecommunication the “Morse Code” was used to transmit texts, using long and short pulses of electricity.
- ▶ **Definition 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 2.8** The Morse Code in the table above induces a character code $\mu: \mathcal{R} \rightarrow \{., -\}$.

Codes on Strings

- ▶ **Definition 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 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)

Prefix Codes induce Codes on Strings

► **Theorem 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.

Sufficient Conditions for Prefix Codes

► **Theorem 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 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 2.14** The new character that makes an arbitrary code a prefix code in the construction of Theorem 2.13 is often called a **stop character**.

7.3 Character Codes in the Real World

The ASCII Character Code

- ▶ **Definition 3.1** The **American Standard Code for Information Interchange** (ASCII) is a character code that assigns characters to numbers 0-127

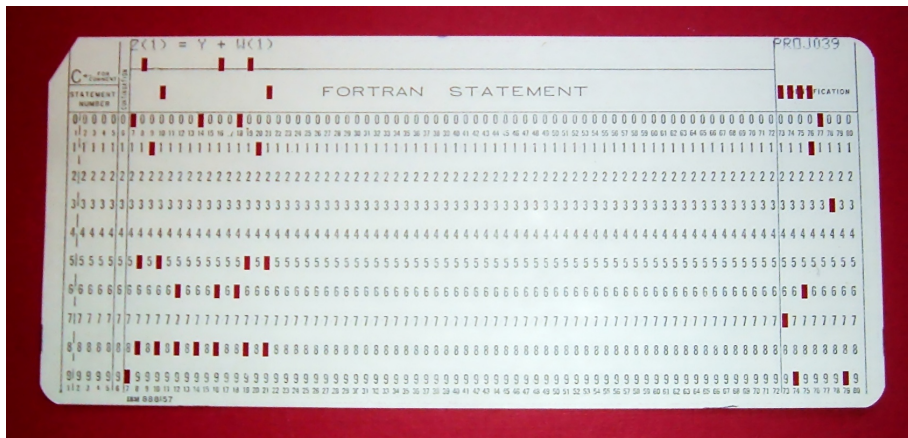
Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	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)

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



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

Unicode and the Universal Character Set

- ▶ **Definition 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 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 3.7** Most (non-Chinese) characters have code points in [1, 65536] (the **basic multilingual plane**).
- ▶ **Notation 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;

Character Encodings in Unicode


- ▶ **Definition 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 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 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 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)

7.4 Formal Languages and Meaning

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 names**,
 - ▶ $V := \{x\} \times \{1, \dots, 9\} \times \{0, \dots, 9\}^*$ is a set of **variable names**,
 - ▶ $F_{\text{un}}^2 := \{\text{add}, \text{sub}, \text{mul}, \text{div}, \text{mod}\}$ is a set of (binary) **function names**, and
 - ▶ $B := \{(\, ,)\} \cup \{, \}$ is a set of **structural characters**. ( “,” (“,”) characters!)
 - ▶ define strings in stages: $E_{\text{un}} := \bigcup_{i \in \mathbb{N}} E_{\text{un}}^i$, 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.

A formal Language for Unary Arithmetics (Examples)

► **Example 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!)

Syntax and Semantics (a first glimpse)

- ▶ **Definition 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. $\text{add}(//////, \text{mul}(x_{1902}, ///)) \mapsto 6 + (x_{1902} \cdot 3)$ (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 $\text{add}(//////, \text{mul}(x_{1902}, ///)) \mapsto 6 + (3 \cdot 3) = 15$

Chapter 8 Midterm Analysis

Mid-Term Results

- ▶ **Midterm:** 9 Problems, 55 Points,
- ▶ **Grades:** Average grade: 3.33 (barely “satisfactory”)

Performance	exc.		very good		good		satisfactory			sufficient		failing		
Jacobs Grade	1.00	1.33	1.67	2.00	2.33	2.67	3.00	3.33	3.67	4.00	4.33	4.67	5.00	∅
Cardinality	14	2	1	4	2	3	5	8	5	5	4	4	21	6
	█	█	█	█	█	█	█	█	█	█	█	█	█	█

- ▶ **simplest problems:** Greek letters (avg. 77%), function definition (avg. 64%),
- ▶ **hardest problems:** CNF/DNF (avg. 14%), SML (avg. 36%)

Procedure, Consequences

- ▶ Procedure (some order please to minimize this boring but necessary task)
 - ▶ Go to Exam Inspection (Today! 13:00-14:00 GenCS Lounge (103@R1))
 - ▶ You will check the grading, points summation, ...
 - ▶ We will answer questions, and correct mistakes.
 - ▶ You will take home the test, when you leave the room the grade is final!
- ▶ Consequences (we all want a better result in the final and the final grade)
 - ▶ try to know actively (just passively understanding is not enough)
 - ▶ try to write anything at all (so we can give you partial points)
 - ▶ you will need to take more advantage of tutorials and TAs (we are here to help you!)
- ▶ There is really no need to fail this course (if you do, rethink your major)

Chapter 9 Boolean Algebra

9.1 Boolean Expressions and their Meaning

Let us try again (Boolean Expressions)

- ▶ **Definition 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 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 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)

Boolean Expressions as Structured Objects.

- ▶ **Idea:** As strings in E_{bool} are built up via the “union-principle”, we can think of them as constructor terms with variables
- ▶ **Definition 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 1.5** $\sigma : E_{\text{bool}} \rightarrow \mathcal{T}_{\mathbb{B}}(\mathcal{B}; \mathcal{V})$ defined by


$$\sigma(1) := 1 \qquad \sigma(0) := 0$$

$$\sigma((-A)) := (-\sigma(A))$$

$$\sigma((A*B)) := (\sigma(A)*\sigma(B)) \quad \sigma((A+B)) := (\sigma(A)+\sigma(B))$$

- ▶ 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 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 1.7** $\sigma(((-(x_1*x_2)))+(x_3*x_4))) = \overline{x_1 * x_2} + x_3 * x_4$

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

Boolean Expressions: Semantics via Models

- ▶ **Definition 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 1.9** A function $\varphi: V \rightarrow \mathcal{U}$ is called a **variable assignment**.
- ▶ **Definition 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
 - ▶ $\mathcal{I}_\varphi(c) = \mathcal{I}(c)$, for $c \in C_{\text{bool}}$
 - ▶ $\mathcal{I}_\varphi(x) = \varphi(x)$, 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 $\mathcal{I}_\varphi(a * b) = \mathcal{I}(*)(\mathcal{I}_\varphi(a), \mathcal{I}_\varphi(b))$
- ▶ $\mathcal{U} = \{T, F\}$ with $0 \mapsto F, 1 \mapsto T, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg$.
- ▶ $\mathcal{U} = E_{\text{un}}$ with $0 \mapsto /, 1 \mapsto //, + \mapsto \text{div}, * \mapsto \text{mod}, - \mapsto \lambda x.(///)$.
- ▶ $\mathcal{U} = \{0, 1\}$ with $0 \mapsto 0, 1 \mapsto 1, + \mapsto \min, * \mapsto \max, - \mapsto \lambda x.1 - x$.

Evaluating Boolean Expressions

- **Example 1.11** Let $\varphi := [T/x_1], [F/x_2], [T/x_3], [F/x_4]$, and $\mathcal{I} = \{0 \mapsto F, 1 \mapsto T, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\}$, then

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

- What a mess!

Boolean Algebra

- ▶ **Definition 1.12** A **Boolean algebra** is E_{bool} together with the models
 - ▶ $\langle \{\text{T}, \text{F}\}, \{0 \mapsto \text{F}, 1 \mapsto \text{T}, + \mapsto \vee, * \mapsto \wedge, - \mapsto \neg\} \rangle$.
 - ▶ $\langle \{0, 1\}, \{0 \mapsto 0, 1 \mapsto 1, + \mapsto \max, * \mapsto \min, - \mapsto \lambda x.1 - x\} \rangle$.
- ▶ BTW, the models are equivalent $(0 \hat{=} \text{F}, 1 \hat{=} \text{T})$
- ▶ **Definition 1.13** We will use \mathbb{B} for the universe, which can be either $\{0, 1\}$ or $\{\text{T}, \text{F}\}$
- ▶ **Definition 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 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 .

A better mouse-trap: Truth Tables

- ▶ Truth tables to visualize truth functions:

$\bar{\cdot}$		$*$	T	F	$+$	T	F
T	F	T	T	F	T	T	T
F	T	F	F	F	F	T	F

- ▶ If we are interested in values for all assignments (e.g. of $x_{123} * x_4 + \overline{x_{123}} * x_72$)

assignments			intermediate results			full
x_4	x_72	x_{123}	$e_1 := x_{123} * x_72$	$e_2 := \overline{e_1}$	$e_3 := x_{123} * x_4$	$e_3 + e_2$
F	F	F	F	T	F	T
F	F	T	F	T	F	T
F	T	F	F	T	F	T
F	T	T	T	F	F	F
T	F	F	F	T	F	T
T	F	T	F	T	T	T
T	T	F	F	T	F	T
T	T	T	T	F	T	T

Boolean Equivalences

- ▶ Given $a, b, c \in E_{\text{bool}}$, $\circ \in \{+, *\}$, 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)
 - ▶ $(a \circ b) \hat{\circ} (a \circ \bar{b}) \equiv a$ (combining)
 - ▶ $(a \circ b) \hat{\circ} ((\bar{a} \circ c) \hat{\circ} (b \circ c)) \equiv (a \circ b) \hat{\circ} (\bar{a} \circ c)$ (consensus)
 - ▶ $\overline{a \circ b} \equiv \bar{a} \hat{\circ} \bar{b}$ (De Morgan)

9.2 Boolean Functions

Boolean Functions

- ▶ **Definition 2.1** A **Boolean function** is a function from \mathbb{B}^n to \mathbb{B} .
- ▶ **Definition 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 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 $x_i \prec_{\text{lex}} x_j$ where $i \leq j$ and \prec is the “numerical order” on variables.
- ▶ **Definition 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 2.5** $e_1 \equiv e_2$, iff $f_{e_1} = f_{e_2}$.

Boolean Functions and Truth Tables


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

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

- ▶ compute this by assigning values and evaluating
- ▶ **Question:** can we also go the other way? (from function to expression?)
- ▶ **Idea:** read expression of a special form from truth tables (Boolean Polynomials)

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 2.6** Write $x_i x_j$ instead of $x_i * x_j$. (like in math)

Normal Forms of Boolean Functions

- ▶ **Definition 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 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 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 : $(x_1 + \overline{x_2} + x_3)(x_1 + \overline{x_2} + \overline{x_3})(\overline{x_1} + \overline{x_2} + x_3)$

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 2.10** The **cost** $C(e)$ of $e \in E_{\text{bool}}$ is the number of operators in e .
- ▶ **Example 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 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 ?

9.3 Complexity Analysis for Boolean Expressions

9.3.1 The Mathematics of Complexity

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

▶ **Definition 3.1** 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 3.2** The three **Landau sets** $O(g), \Omega(g), \Theta(g)$ are defined as

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

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

▶ $\Theta(g) = O(g) \cap \Omega(g)$

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

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

▶ **Notation 3.3** (\triangleleft) We often see $f = O(g)$ as a statement of complexity; this is a funny notation for $n \in f O(g)$!

Computing with Landau Sets

- ▶ **Lemma 3.4** *We have the following computation rules for Landau sets:*
 - ▶ If $k \neq 0$ and $f \in O(g)$, then $(k f) \in O(g)$.
 - ▶ 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 3.5** 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 3.6** $(\lambda n.263748) \in O(1)$
- ▶ **Example 3.7** $(\lambda n.26n + 372) \in O(n)$
- ▶ **Example 3.8** $(\lambda n.857n^{10} + 7342n^7 + 26n^2 + 902) \in O(n^{10})$
- ▶ **Example 3.9** $(\lambda n.3 \cdot 2^n + 72) \in O(2^n)$
- ▶ **Example 3.10** $(\lambda n.3 \cdot 2^n + 7342n^7 + 26n^2 + 722) \in O(2^n)$

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 3.11** 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

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

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

9.3.2 Asymptotic Bounds for Costs of Boolean Expressions

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 3.12** 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	$x_1^0 x_2^0 x_3^0$	$x_1^1 + x_2^0 + x_3^1$
0	0	1	1	$x_1^0 x_2^0 x_3^1$	
0	1	0	0		
0	1	1	0		$x_1^1 + x_2^0 + x_3^0$
1	0	0	1	$x_1^1 x_2^0 x_3^0$	$x_1^0 + x_2^0 + x_3^1$
1	0	1	1	$x_1^1 x_2^0 x_3^1$	
1	1	0	0		
1	1	1	1	$x_1^1 x_2^1 x_3^1$	

- ▶ **Theorem 3.13** Any $f : \mathbb{B}^n \rightarrow \mathbb{B}$ is realized by an $e \in E_{\text{bool}}$ 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 $C(e_n) \in O(n \cdot 2^n)$ □

We can do better (if we accept complicated witness)

▶ **Theorem 3.14** 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 $\kappa(1) = 4$ 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 $2 \leq n$. □

□

A Lower Bound for the Cost of BF with n Variables

► **Theorem 3.15** $\kappa \in \Omega\left(\frac{2^n}{\log_2(n)}\right)$

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

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

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

▶ **Definition 3.16** $\mathcal{E}_n := \{e \in E_{\text{bool}} \mid e \text{ has } n \text{ variables and } C(e) \leq \kappa(n)\}$

▶ **Lemma 3.17** $\#(\mathcal{E}_n) \geq \#(\mathbb{B}^n \rightarrow \mathbb{B})$

▶ **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 $g = h$. □

▶ **Corollary 3.18** $\#(\mathcal{E}_n) \geq 2^{2^n}$

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

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 3.19** 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 3.20** $\sigma(n) \leq 5n$ 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

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

- ▶ **Corollary 3.21** $\max\{\Lambda(e) \mid e \in \mathcal{E}_n\} \leq 5 \cdot \kappa(n)$

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

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



Solving for $\kappa(n)$

- ▶ $\frac{(n+7)^{5\kappa(n)+1}-1}{n+6} \geq 2^{2^n}$
- ▶ $(n+7)^{5\kappa(n)+1} \geq 2^{2^n}$
- ▶ $5\kappa(n) + 1 \cdot \log_2(n+7) \geq 2^n$
- ▶ $5\kappa(n) + 1 \geq \frac{2^n}{\log_2(n+7)}$
- ▶ $\kappa(n) \geq 1/5 \cdot \frac{2^n}{\log_2(n+7)} - 1$
- ▶ $\kappa(n) \in \Omega\left(\frac{2^n}{\log_2(n)}\right)$

$$\begin{aligned} & \text{(as } (n+7)^{5\kappa(n)+1} \geq \frac{(n+7)^{5\kappa(n)+1}-1}{n+6} \text{)} \\ & \text{(as } \log_a(x) = \log_b(x) \cdot \log_a(b) \text{)} \end{aligned}$$

9.4 The Quine-McCluskey Algorithm

Constructing Minimal Polynomials: Prime Implicants

- ▶ **Definition 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 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 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' .

Constructing Minimal Polynomials: Prime Implicants

► **Lemma 4.4** *If $M' \subset M$, then M' dominates M .*

► **Proof:**

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

P.2 As M' is a sub-monomial of M , then $f_{L'_j}(c) = \top$ for each literal L'_j of M' .

P.3 Therefore, $f_{M'}(c) = \top$. □

► **Definition 4.5** An implicant M of f is a **prime implicant** of f iff no sub-monomial of M is an implicant of f .

Prime Implicants and Costs

- ▶ **Theorem 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 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. □

The Quine/McCluskey Algorithm (Idea)

- ▶ **Idea:** use Theorem 4.6 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: $p \equiv m$)

The algorithm QMC₁, for determining Prime Implicants

- ▶ **Definition 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 **resolvents** of M
 - ▶ $\widehat{\mathcal{R}}(M) := \{m \in M \mid m \text{ has a partner in } M\}$ ($n\bar{x}_i$ and nx_i are partners)
- ▶ **Definition 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)
 - ▶ $P_j := M_{j-1} \setminus \widehat{\mathcal{R}}(M_{j-1})$ (get rid of redundant resolution partners)
 - ▶ terminate when $M_j = \emptyset$, return $P_{\text{prime}} := \bigcup_{j=1}^n P_j$

Example for QMC₁

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

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

$$M_0 = \{ \underbrace{\overline{x1 x2 x3}}_{=: e_1^0}, \underbrace{\overline{x1 x2 x3}}_{=: e_2^0}, \underbrace{\overline{x1 x2 x3}}_{=: e_3^0}, \underbrace{\overline{x1 x2 x3}}_{=: e_4^0}, \underbrace{\overline{x1 x2 x3}}_{=: e_5^0} \}$$

$$M_1 = \{ \underbrace{\overline{x1 x2}}_{\mathcal{R}(e_1^0, e_2^0) =: e_1^1}, \underbrace{\overline{x2 x3}}_{\mathcal{R}(e_1^0, e_3^0) =: e_2^1}, \underbrace{\overline{x2 x3}}_{\mathcal{R}(e_2^0, e_4^0) =: e_3^1}, \underbrace{\overline{x1 x2}}_{\mathcal{R}(e_3^0, e_4^0) =: e_4^1}, \underbrace{\overline{x1 x3}}_{\mathcal{R}(e_4^0, e_5^0) =: e_5^1} \}$$

$$P_1 = \emptyset$$

$$M_2 = \{ \underbrace{\overline{x2}}_{\mathcal{R}(e_1^1, e_4^1) =: e_1^2}, \underbrace{\overline{x2}}_{\mathcal{R}(e_2^1, e_3^1) =: e_2^2} \}$$

$$P_2 = \{x1 x3\}$$

$$M_3 = \emptyset$$

$$P_3 = \{\overline{x2}\}$$

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

- ▶ **Lemma 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 4.10** *QMC₁ terminates after at most n rounds.*
- ▶ **Corollary 4.11** *P_{prime} is the set of all prime implicants of f .*

Algorithm QMC₂: Minimize Prime Implicants Polynomial

- ▶ **Definition 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 4.13** $P_{prime} = \{x_1 x_3, \overline{x_2}\}$, so $e_p \in \{1, x_1 x_3, \overline{x_2}, x_1 x_3 + \overline{x_2}\}$.
- ▶ Only $f_{x_1 x_3 + \overline{x_2}} \equiv f$, so $x_1 x_3 + \overline{x_2}$ is the minimal polynomial
- ▶ **Complaint:** The set of combinations (power set) grows exponentially

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

- ▶ **Definition 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 4.15**

	FFF	FFT	TFF	TFT	TTT
$x_1 x_3$	F	F	F	T	T
$\overline{x_2}$	T	T	T	T	F

- ▶ **Definition 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.

Essential Prime Implicants and Minimal Polynomials

- ▶ **Theorem 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

□

A complex Example for QMC (Function and DNF)

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

A complex Example for QMC (QMC₁)

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

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

$$P_1 = \emptyset$$

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

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

$$M_3 = \emptyset$$

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

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

A better Mouse-trap for QMC₁: 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 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	$x1^0 x2^0 x3^0 x4^0$
F	F	F	T	$x1^0 x2^0 x3^0 x4^1$
F	F	T	F	$x1^0 x2^0 x3^1 x4^0$
F	T	F	T	$x1^0 x2^1 x3^0 x4^1$
T	F	T	F	$x1^1 x2^0 x3^1 x4^0$
T	F	T	T	$x1^1 x2^0 x3^1 x4^1$
T	T	T	F	$x1^1 x2^1 x3^1 x4^0$
T	T	T	T	$x1^1 x2^1 x3^1 x4^1$

~

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

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

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

- ▶ This table represents the prime implicants of f

A complex Example for QMC₁

- ▶ The PIT:

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

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

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

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

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

- ▶ all the remaining ones ($\overline{x1} \overline{x2} x4$, $\overline{x1} x3 x4$, and $x1 x3$) are essential
- ▶ So, the minimal polynomial of f is $\overline{x1} \overline{x2} x4 + \overline{x1} x3 x4 + x1 x3$.

The Quine-McCluskey Algorithm (final version)

- ▶ We started from a simple idea suggested by Theorem 4.6
 - ▶ Determine all prime implicants (sub-algorithm QMC₁)
 - ▶ choose the minimal subset that covers f (sub-algorithm QMC₂)and optimized the parts (data structures and partial algorithms) considerably.
- ▶ **Definition 4.19** The **Quine-McCluskey algorithm** () computes minimal polynomials for a given Boolean function f by computing the set of prime implicants and choosing a covering subset.
- ▶ **Observation**: Good algorithms are often based on mathematical insights (theorems), but math is not enough, considerable work goes into finding good representations (data structures) and clever sub-algorithms.

Chapter 10 Propositional Logic

10.1 Boolean Expressions and Propositional Logic

Still another Notation for Boolean Expressions

- ▶ **Idea:** get closer to MathTalk
 - ▶ Use \vee , \wedge , \neg , \Rightarrow , and \Leftrightarrow directly (after all, we do in MathTalk)
 - ▶ construct more complex names (**propositions**) for variables (Use ground terms of sort \mathbb{B} in an ADT)
- ▶ **Definition 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 1.2** Call terms in $\mathcal{T}_{\mathbb{B}}^g(\Sigma)$ without \vee , \wedge , \neg , \Rightarrow , and \Leftrightarrow **atoms**. (write $\mathcal{A}(\Sigma)$)
- ▶ **Note:** Formulae of propositional logic “are” Boolean Expressions
 - ▶ replace $\mathbf{A} \Leftrightarrow \mathbf{B}$ by $(\mathbf{A} \Rightarrow \mathbf{B}) \wedge (\mathbf{B} \Rightarrow \mathbf{A})$ and $\mathbf{A} \Rightarrow \mathbf{B}$ by $\neg \mathbf{A} \vee \mathbf{B} \dots$
 - ▶ 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 \cdot stands for a left bracket whose partner is as far right as is consistent with existing brackets $(\mathbf{A} \Rightarrow \cdot \mathbf{C} \wedge \mathbf{D} = \mathbf{A} \Rightarrow (\mathbf{C} \wedge \mathbf{D}))$

Semantic Properties of Boolean Expressions

- ▶ **Definition 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 $\mathcal{I}_\varphi(e) = \top$ (write $\mathcal{M} \models^\varphi e$)
 - ▶ **false under φ** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \text{F}$ (write $\mathcal{M} \not\models^\varphi e$)
 - ▶ **satisfiable** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \top$ 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 $\mathcal{I}_\varphi(e) = \text{F}$ for some assignments φ
 - ▶ **unsatisfiable** in \mathcal{M} , iff $\mathcal{I}_\varphi(e) = \text{F}$ for all assignments φ
- ▶ **Example 1.4** $x \vee x$ is satisfiable and falsifiable.
- ▶ **Example 1.5** $x \vee \neg x$ is valid and $x \wedge \neg x$ is unsatisfiable.
- ▶ **Notation 1.6** (alternative) Write $\llbracket e \rrbracket_\varphi^{\mathcal{M}}$ for $\mathcal{I}_\varphi(e)$, if $\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$. (and $\llbracket e \rrbracket^{\mathcal{M}}$, if e is ground, and $\llbracket e \rrbracket$, if \mathcal{M} is clear)
- ▶ **Definition 1.7 (Entailment)** (aka. logical consequence)
We say that e **entails** f ($e \models f$), iff $\mathcal{I}_\varphi(f) = \top$ for all φ with $\mathcal{I}_\varphi(e) = \top$ (i.e. all assignments that make e true also make f true)

Example: Propositional Logic with ADT variables

- ▶ **Idea:** We use propositional logic to express things about the world (PLNQ $\hat{=}$ Predicate Logic without Quantifiers)
- ▶ **Example 1.8** $\mathcal{A} := \langle \{\mathbb{B}, \mathbb{I}\}, \{\dots, [\text{love}: \mathbb{I} \times \mathbb{I} \rightarrow \mathbb{B}], [\text{bill}: \mathbb{I}], [\text{mary}: \mathbb{I}], \dots\} \rangle$
The abstract data type \mathcal{A} has the 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 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 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 $\llbracket \mathbf{A} \rrbracket^{\mathcal{M}} = \top$ (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?

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** := $(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R$

- ▶ $\frac{\mathbf{A} \Rightarrow \mathbf{B} \quad \mathbf{A}}{\mathbf{B}}$ MP
- ▶ $\frac{\mathbf{A}}{[\mathbf{B}/\mathbf{X}](\mathbf{A})}$ Subst

- ▶ Let us look at a \mathcal{H}^0 theorem (with a proof)
- ▶ $\mathbf{C} \Rightarrow \mathbf{C}$ (*Tertium non datur*)

- ▶ **Proof:**

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

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

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

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

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

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



10.2 Calculi for Propositional Logic

Derivation Systems and Inference Rules

- ▶ **Definition 2.1** 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 2.2** 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 2.3** Let \mathcal{L} be a formal language, then an **inference rule** over \mathcal{L}

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

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

- ▶ **Definition 2.4** An inference rule without assumptions is called an **axiom** (schema).
- ▶ **Definition 2.5** 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} .

Derivations and Proofs I

- ▶ **Definition 2.6** 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}_i .

- ▶ **Example 2.7** 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.

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

- ▶ **Observation 2.8** Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{K}, \models \rangle$ be a logical system and \mathcal{C} a calculus for \mathcal{S} , then the \mathcal{C} -derivation relation $\vdash_{\mathcal{D}}$ defined in Definition 2.6 is a derivation relation in the sense of Definition 2.1.¹

Derivations and Proofs II

- ▶ **Definition 2.9** 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 2.10** 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 2.11** 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.

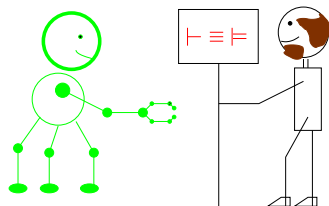
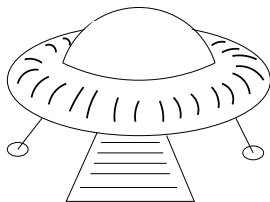
¹EdNote: MK: this should become a view!

Soundness and Completeness

- ▶ **Definition 2.12** 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 $\models \mathbf{A}$
 - ▶ **To TRUTH through PROOF**

(provability and validity coincide)

(CALCULEMUS [Leibniz ~1680])



The miracle of logics

- Purely formal derivations are true in the real world!

World of Logics

$\forall x (\text{human } x \rightarrow \text{mortal } x)$



\wedge

human Socrates

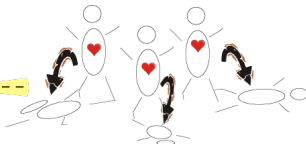


\Downarrow

mortal Socrates



Real World



it's true!

10.3 Proof Theory for the Hilbert Calculus

\mathcal{H}^0 is sound (first version)

► **Theorem 3.1** $\vdash \mathbf{A}$ implies $\models \mathbf{A}$ for all propositions \mathbf{A} .

► **Proof:** show by induction over proof length

P.1 Axioms are valid

(we already know how to do this!)

P.2 inference rules preserve validity

(let's think)

P.2.1 **Subst:** complicated, see next slide

P.2.2 **MP:**

P.2.2.1 Let $\mathbf{A} \Rightarrow \mathbf{B}$ be valid, and $\varphi: \mathcal{V}_o \rightarrow \{\mathbf{T}, \mathbf{F}\}$ arbitrary

P.2.2.2 then $\mathcal{I}_\varphi(\mathbf{A}) = \mathbf{F}$ or $\mathcal{I}_\varphi(\mathbf{B}) = \mathbf{T}$ (by definition of \Rightarrow).

P.2.2.3 Since \mathbf{A} is valid, $\mathcal{I}_\varphi(\mathbf{A}) = \mathbf{T} \neq \mathbf{F}$, so $\mathcal{I}_\varphi(\mathbf{B}) = \mathbf{T}$.

P.2.2.4 As φ was arbitrary, \mathbf{B} is valid.



\mathcal{H}^0 axioms are valid

► **Lemma 3.2** *The \mathcal{H}^0 axioms are valid.*

► **Proof:** We simply check the truth tables

P.1

P	Q	$Q \Rightarrow P$	$P \Rightarrow Q \Rightarrow P$
F	F	T	T
F	T	F	T
T	F	T	T
T	T	T	T

P.2

P	Q	R	$A := P \Rightarrow Q \Rightarrow R$	$B := P \Rightarrow Q$	$C := P \Rightarrow R$	$A \Rightarrow B \Rightarrow C$
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



Substitution Value Lemma and Soundness

- ▶ **Lemma 3.3** 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 $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \text{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{C})) = \text{F}$ or $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{D})) = \text{T}$.
 - P.1.2.2 This is the case, iff $\mathcal{I}_\psi(\mathbf{C}) = \text{F}$ or $\mathcal{I}_\psi(\mathbf{D}) = \text{T}$ by IH (\mathbf{C} and \mathbf{D} have smaller depth than \mathbf{A}).
 - P.1.2.3 In other words, $\mathcal{I}_\psi(\mathbf{A}) = \mathcal{I}_\psi(\mathbf{C} \Rightarrow \mathbf{D}) = \text{T}$, iff $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \text{T}$ by definition.
 - P.2 We have considered all the cases and proven the assertion.

Soundness of Substitution

► **Lemma 3.4** *Subst preserves validity.*

► **Proof:** We have to show that $[\mathbf{B}/X](\mathbf{A})$ is valid, if \mathbf{A} is.

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

P.2 then $\mathcal{I}_\varphi([\mathbf{B}/X](\mathbf{A})) = \mathcal{I}_{\varphi, [\mathcal{I}_\varphi(\mathbf{B})/X]}(\mathbf{A}) = \top$, since \mathbf{A} is valid.

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

The Entailment Theorem

- ▶ **Theorem 3.5** If $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, then $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.
- ▶ **Proof:** We show that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \top$ for all assignments φ with $\mathcal{I}_\varphi(\mathcal{H}) = \top$ whenever $\mathcal{H}, \mathbf{A} \models \mathbf{B}$
 - P.1 Let us assume there is an assignment φ , such that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \text{F}$.
 - P.2 Then $\mathcal{I}_\varphi(\mathbf{A}) = \top$ and $\mathcal{I}_\varphi(\mathbf{B}) = \text{F}$ by definition.
 - P.3 But we also know that $\mathcal{I}_\varphi(\mathcal{H}) = \top$ and thus $\mathcal{I}_\varphi(\mathbf{B}) = \top$, since $\mathcal{H}, \mathbf{A} \models \mathbf{B}$.
 - P.4 This contradicts our assumption $\mathcal{I}_\varphi(\mathbf{B}) = \text{F}$ from above.
 - P.5 So there cannot be an assignment φ that $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \text{F}$; in other words, $\mathbf{A} \Rightarrow \mathbf{B}$ is valid. □

The Entailment Theorem (continued)

- ▶ **Corollary 3.6** $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, iff $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$
- ▶ **Proof:** In the light of the previous result, we only need to prove that $\mathcal{H}, \mathbf{A} \models \mathbf{B}$, whenever $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$
 - P.1 To prove that $\mathcal{H}, \mathbf{A} \models \mathbf{B}$ we assume that $\mathcal{I}_\varphi(\mathcal{H}, \mathbf{A}) = \top$.
 - P.2 In particular, $\mathcal{I}_\varphi(\mathbf{A} \Rightarrow \mathbf{B}) = \top$ since $\mathcal{H} \models (\mathbf{A} \Rightarrow \mathbf{B})$.
 - P.3 Thus we have $\mathcal{I}_\varphi(\mathbf{A}) = \top$ or $\mathcal{I}_\varphi(\mathbf{B}) = \top$.
 - P.4 The first cannot hold, so the second does, thus $\mathcal{H}, \mathbf{A} \models \mathbf{B}$. □

The Deduction Theorem I

► **Theorem 3.7** 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.3.12 **Case 1:** \mathbf{C}_i axiom or $\mathbf{C}_i \in \mathcal{H}$:

P.3.12.1 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.3.12.2 So $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP. □

P.3.13 **Case 2:** $\mathbf{C}_i = \mathbf{A}$:

P.3.13.1 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) □

The Deduction Theorem II

P.3.14 Case 3: *everything else*:

P.3.14.1 \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.3.14.2 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.3.14.3 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.3.14.4 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 $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{B}$. □

The missing Subst case

- ▶ **Oooops:** The proof of the deduction theorem was incomplete (we did not treat the Subst case)
- ▶ **Let's try:**
- ▶ **Proof:** \mathbf{C}_i is inferred by Subst from \mathbf{C}_j for $j < i$ with $[\mathbf{B}/\mathbf{X}]$.
 - P.1 So $\mathbf{C}_i = [\mathbf{B}/\mathbf{X}](\mathbf{C}_j)$; we have $\mathcal{H} \vdash \mathbf{A} \Rightarrow \mathbf{C}_j$ by IH
 - P.2 so by Subst we have $\mathcal{H} \vdash [\mathbf{B}/\mathbf{X}](\mathbf{A} \Rightarrow \mathbf{C}_j)$. (Oooops! $\neq \mathbf{A} \Rightarrow \mathbf{C}_i$)

□

Repairing the Subst case by repairing the calculus

- ▶ **Idea:** Apply Subst only to axioms (this was sufficient in our example)
- ▶ \mathcal{H}^1 Axiom Schemata: (infinitely many axioms)
 $\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}, \quad (\mathbf{A} \Rightarrow \mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}$
Only one inference rule: MP.
- ▶ **Definition 3.8** \mathcal{H}^1 introduces a (potentially) different derivability relation than \mathcal{H}^0 we call them $\vdash_{\mathcal{H}^0}$ and $\vdash_{\mathcal{H}^1}$

Deduction Theorem Redone

► **Theorem 3.9** If $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^1} \mathbf{B}$, then $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{B}$

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

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

P.2 We have to consider three cases

P.2.1 \mathbf{C}_i is an axiom or hypothesis:

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

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

P.2.2 $\mathbf{C}_i = \mathbf{A}$:

P.2.2.1 We have proven $\emptyset \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{A}$, (check proof in \mathcal{H}^1)

We have $\emptyset \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$, so in particular $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ □

P.2.3 else:

P.2.3.1 \mathbf{C}_i is inferred by MP from \mathbf{C}_j and $\mathbf{C}_k = \mathbf{C}_j \Rightarrow \mathbf{C}_i$ for $j, k < i$

P.2.3.2 We have $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j$ and $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i$ by IH

P.2.3.3 Furthermore, $(\mathbf{A} \Rightarrow \mathbf{C}_j \Rightarrow \mathbf{C}_i) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}_j) \Rightarrow \mathbf{A} \Rightarrow \mathbf{C}_i$ by Axiom 2

P.2.3.4 and thus $\mathcal{H} \vdash_{\mathcal{H}^1} \mathbf{A} \Rightarrow \mathbf{C}_i$ by MP (twice). (no Subst) □

□

□

The Deduction Theorem for \mathcal{H}^0

► **Lemma 3.10** $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$

► **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. □

► **Corollary 3.11** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$, iff $\mathcal{H} \vdash_{\mathcal{H}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

► **Proof Sketch:** by MP and $\vdash_{\mathcal{H}^1} = \vdash_{\mathcal{H}^0}$ □

\mathcal{H}^0 is sound (full version)

► **Theorem 3.12** For all propositions \mathbf{A} , \mathbf{B} , we have $\mathbf{A} \vdash_{\mathcal{H}^0} \mathbf{B}$ implies $\mathbf{A} \models \mathbf{B}$.

► **Proof:**

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

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

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

Properties of Calculi (Theoretical Logic)

▶ **Correctness:**

▶ $\mathcal{H} \vdash \mathbf{B}$ implies $\mathcal{H} \models \mathbf{B}$

▶ **Completeness:**

▶ $\mathcal{H} \models \mathbf{B}$ implies $\mathcal{H} \vdash \mathbf{B}$

▶ **Goal:** $\vdash \mathbf{A}$ iff $\models \mathbf{A}$

▶ **To TRUTH through PROOF**

(provable implies valid)

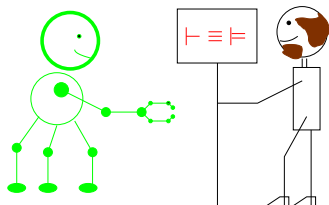
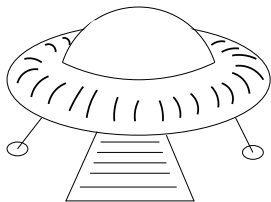
(equivalent: $\vdash \mathbf{A}$ implies $\models \mathbf{A}$)

(valid implies provable)

(equivalent: $\models \mathbf{A}$ implies $\vdash \mathbf{A}$)

(provability and validity coincide)

(CALCULEMUS [Leibniz ~1680])



10.4 A Calculus for Mathtalk

10.4.1 Propositional Natural Deduction Calculus

Calculi: Natural Deduction (\mathcal{ND}^0 ; Gentzen [Gen35])

- ▶ **Idea:** \mathcal{ND}^0 tries to mimic human theorem proving behavior (non-minimal)
- ▶ **Definition 4.1** The **propositional natural deduction calculus** \mathcal{ND}^0 has rules for the introduction and elimination of connectives

Introduction

$$\frac{A \quad B}{A \wedge B} \wedge I$$

Elimination

$$\frac{A \wedge B}{A} \wedge E_l \quad \frac{A \wedge B}{B} \wedge E_r$$

Axiom

$$\frac{}{A \vee \neg A} \text{TND}$$

$[A]^1$

$$\frac{\frac{}{B}}{A \Rightarrow B} \Rightarrow I^1$$

$$\frac{A \Rightarrow B \quad A}{B} \Rightarrow E$$

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

Natural Deduction: Examples

- Inference with local hypotheses

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

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

A Deduction Theorem for \mathcal{ND}^0

► **Theorem 4.2** $\mathcal{H}, \mathbf{A} \vdash_{\mathcal{ND}^0} \mathbf{B}$, iff $\mathcal{H} \vdash_{\mathcal{ND}^0} \mathbf{A} \Rightarrow \mathbf{B}$.

► **Proof:** We show the two directions separately

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

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

More Rules for Natural Deduction

- **Definition 4.3** \mathcal{ND}^0 has the following additional rules for the remaining connectives.

$$\begin{array}{c}
 \frac{A}{A \vee B} \vee I_l \quad \frac{B}{A \vee B} \vee I_r \quad \frac{A \vee B \quad \begin{array}{c} [A]^1 \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B]^1 \\ \vdots \\ C \end{array}}{C} \vee E^1 \\
 \\
 \frac{\begin{array}{c} [A]^1 \\ \vdots \\ F \end{array}}{\neg A} \neg I^1 \quad \frac{\neg \neg A}{A} \neg E \\
 \\
 \frac{\neg A \quad A}{F} FI \quad \frac{F}{A} FE
 \end{array}$$

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

- ▶ Rules for propositional connectives just as always
- ▶ **Definition 4.4 (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.

Natural Deduction with Equality

- ▶ **Definition 4.5 (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 $\mathcal{I}(=) := \{(x, x) \mid x \in \mathcal{D}_i\}$. We call the extended logic **first-order logic with equality** ($PL_{=}^1$)
- ▶ We now extend natural deduction as well.
- ▶ **Definition 4.6** 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} .

Chapter 11 Machine-Oriented Calculi

11.1 Calculi for Automated Theorem Proving: Analytical Tableaux

11.1.1 Analytical Tableaux

Recap: Atoms and Literals

- ▶ **Definition 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 1.2** We call a pair \mathbf{A}^α a **labeled formula**, if $\alpha \in \{\top, \text{F}\}$. A labeled atom is called **literal**.
- ▶ **Definition 1.3** Let Φ be a set of formulae, then we use $\Phi^\alpha := \{\mathbf{A}^\alpha \mid \mathbf{A} \in \Phi\}$.

Test Calculi: Tableaux and Model Generation

- ▶ **Idea:** instead of showing $\emptyset \vdash Th$, show $\neg Th \vdash \text{trouble}$ (use \perp for trouble)
- ▶ **Example 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 \quad \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 \quad 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)

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

$$\begin{array}{c} \mathbf{A} \wedge \mathbf{B}^T \\ \mathbf{A}^T \\ \mathbf{B}^T \end{array} \mathcal{T}_0 \wedge \qquad \begin{array}{c} \mathbf{A} \wedge \mathbf{B}^F \\ \mathbf{A}^F \mid \mathbf{B}^F \end{array} \mathcal{T}_0 \vee \qquad \begin{array}{c} \neg \mathbf{A}^T \\ \mathbf{A}^F \end{array} \mathcal{T}_0 \neg \qquad \begin{array}{c} \neg \mathbf{A}^F \\ \mathbf{A}^T \end{array} \mathcal{T}_0 \neg \qquad \frac{\mathbf{A}^\alpha \quad \mathbf{A}^\beta \quad \alpha \neq \beta}{\perp} \mathcal{T}_0 \text{cut}$$

- ▶ Use rules exhaustively as long as they contribute new material
- ▶ **Definition 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 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 .

A Valid Real-World Example

- **Example 1.7** *If Mary loves Bill and John loves Mary, then John loves Mary*

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

This is a closed tableau, so the

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

As we will see, \mathcal{T}_0 is sound and complete, so

$\text{love}(\text{mary}, \text{bill}) \wedge \text{love}(\text{john}, \text{mary}) \Rightarrow \text{love}(\text{john}, \text{mary})$ is valid.

Deriving Entailment in \mathcal{T}_0

- **Example 1.8** *Mary loves Bill* and *John loves Mary* together entail that *John loves Mary*

$$\begin{array}{c} \text{love}(\text{mary}, \text{bill})^T \\ \text{love}(\text{john}, \text{mary})^T \\ \text{love}(\text{john}, \text{mary})^F \\ \perp \end{array}$$

This is a closed tableau, so the

$\{\text{love}(\text{mary}, \text{bill}), \text{love}(\text{john}, \text{mary})\} \vdash_{\mathcal{T}_0} \text{love}(\text{john}, \text{mary})$, again, as \mathcal{T}_0 is sound and complete we have $\{\text{love}(\text{mary}, \text{bill}), \text{love}(\text{john}, \text{mary})\} \models \text{love}(\text{john}, \text{mary})$

A Falsifiable Real-World Example

- **Example 1.9** **If Mary loves Bill or John loves Mary, then John loves Mary*
Try proving the implication (this fails)

$$\begin{aligned} & (\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary})) \Rightarrow \text{love}(\text{john}, \text{mary})^{\text{F}} \\ \neg & (\neg \neg (\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary})) \wedge \neg \text{love}(\text{john}, \text{mary}))^{\text{F}} \\ & \neg \neg (\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary})) \wedge \neg \text{love}(\text{john}, \text{mary})^{\text{T}} \\ & \quad \neg \text{love}(\text{john}, \text{mary})^{\text{T}} \\ & \quad \text{love}(\text{john}, \text{mary})^{\text{F}} \\ & \neg \neg (\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary}))^{\text{T}} \\ & \neg (\text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary}))^{\text{F}} \\ & \quad \text{love}(\text{mary}, \text{bill}) \vee \text{love}(\text{john}, \text{mary})^{\text{T}} \\ & \quad \text{love}(\text{mary}, \text{bill})^{\text{T}} \mid \text{love}(\text{john}, \text{mary})^{\text{T}} \\ & \quad \quad \quad \perp \end{aligned}$$

Indeed we can make $\mathcal{I}_\varphi(\text{love}(\text{mary}, \text{bill})) = \text{T}$ but $\mathcal{I}_\varphi(\text{love}(\text{john}, \text{mary})) = \text{F}$.

Testing for Entailment in \mathcal{T}_0

- **Example 1.10** Does *Mary loves Bill or John loves Mary* entail that *John loves Mary*?

$$\begin{array}{c} \text{love(mary, bill)} \vee \text{love(john, mary)}^T \\ \text{love(john, mary)}^F \\ \text{love(mary, bill)}^T \quad | \quad \text{love(john, mary)}^T \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \perp \end{array}$$

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

11.1.2 Practical Enhancements for Tableaux

Propositional Identities

- ▶ **Definition 1.11** Let T and F be new logical constants with $\mathcal{I}(T) = T$ and $\mathcal{I}(F) = F$ for all assignments φ .
- ▶ We have the following identities:

Name	for \wedge	for \vee
Idempotence	$\varphi \wedge \varphi = \varphi$	$\varphi \vee \varphi = \varphi$
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	$\varphi \Rightarrow \psi = \neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Derived Rules of Inference

► **Definition 1.12** Let \mathcal{C} be a calculus, a rule of inference $\frac{\mathbf{A}_1 \ \dots \ \mathbf{A}_n}{\mathbf{C}}$ is called a **derived inference rule** in \mathcal{C} , iff there is a \mathcal{C} -proof of $\mathbf{A}_1, \dots, \mathbf{A}_n \vdash \mathbf{C}$.

► **Definition 1.13** We have the following derived rules of inference

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

Tableaux with derived Rules (example)

Example 1.14

$$\begin{array}{l} \text{love}(\text{mary}, \text{bill}) \wedge \text{love}(\text{john}, \text{mary}) \Rightarrow \text{love}(\text{john}, \text{mary})^F \\ \text{love}(\text{mary}, \text{bill}) \wedge \text{love}(\text{john}, \text{mary})^T \\ \quad \text{love}(\text{john}, \text{mary})^F \\ \quad \text{love}(\text{mary}, \text{bill})^T \\ \quad \text{love}(\text{john}, \text{mary})^T \\ \quad \perp \end{array}$$

11.1.3 Soundness and Termination of Tableaux

Soundness (Tableau)

- ▶ **Idea:** A test calculus is sound, iff it preserves satisfiability and the goal formulae are unsatisfiable.
- ▶ **Definition 1.15** A labeled formula \mathbf{A}^α is valid under φ , iff $\mathcal{I}_\varphi(\mathbf{A}) = \alpha$.
- ▶ **Definition 1.16** 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 1.17** *Tableau rules transform satisfiable tableaux into satisfiable ones.*
- ▶ **Theorem 1.18 (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 1.17.
 - P.3 there is a satisfiable branch (by definition)
 - P.4 but all branches are closed (\mathcal{T} closed)

□

Termination for Tableaux

- ▶ **Lemma 1.19** *The tableau procedure terminates, i.e. after a finite set of rule applications, it reaches a tableau, so that applying the tableau rules will only add labeled formulae that are already present on the branch.*
- ▶ Let us call a labeled formulae \mathbf{A}^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 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. □

11.2 Resolution for Propositional Logic

Another Test Calculus: Resolution

- ▶ **Definition 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 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 2.3 (Resolution Refutation)** Let S be a clause set, and $\mathcal{D}: S \vdash_{\mathcal{R}} T$ a \mathcal{R} derivation then we call \mathcal{D} **resolution refutation**, iff $\square \in T$.

A calculus for CNF Transformation

- ▶ **Definition 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 2.5** We write $\mathit{CNF}(\mathbf{A})$ for the set of all clauses derivable from \mathbf{A}^F via the rules above.
- ▶ **Definition 2.6 (Resolution Proof)** We call a resolution refutation $\mathcal{P}: \mathit{CNF}(\mathbf{A}) \vdash_{\mathcal{R}} T$ a **resolution proof** for $\mathbf{A} \in \mathit{wff}_o(\mathcal{V}_o)$.

Derived Rules of Inference

- **Definition 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 2.8**

$$\frac{\frac{\frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^T}{\mathbf{C} \vee (\neg \mathbf{A} \vee \mathbf{B})^T}}{\mathbf{C} \vee \neg \mathbf{A}^T \vee \mathbf{B}^T}}{\mathbf{C} \vee \mathbf{A}^F \vee \mathbf{B}^T} \mapsto \frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^T}{\mathbf{C} \vee \mathbf{A}^F \vee \mathbf{B}^T}$$

- **Others:**

$$\frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^T}{\mathbf{C} \vee \mathbf{A}^F \vee \mathbf{B}^T} \quad \frac{\mathbf{C} \vee (\mathbf{A} \Rightarrow \mathbf{B})^F}{\mathbf{C} \vee \mathbf{A}^T; \mathbf{C} \vee \mathbf{B}^F} \quad \frac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^T}{\mathbf{C} \vee \mathbf{A}^T; \mathbf{C} \vee \mathbf{B}^T} \quad \frac{\mathbf{C} \vee \mathbf{A} \wedge \mathbf{B}^F}{\mathbf{C} \vee \mathbf{A}^F \vee \mathbf{B}^F}$$

Example: Proving Axiom S

▶ Example 2.9 Clause Normal Form transformation

$$\frac{(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \Rightarrow Q) \Rightarrow P \Rightarrow R^F}{P \Rightarrow Q \Rightarrow R^T; (P \Rightarrow Q) \Rightarrow P \Rightarrow R^F}$$
$$\frac{P^F \vee (Q \Rightarrow R)^T; P \Rightarrow Q^T; P \Rightarrow R^F}{P^F \vee Q^F \vee R^T; P^F \vee Q^T; P^T; R^F}$$

$$CNF = \{P^F \vee Q^F \vee R^T, P^F \vee Q^T, P^T, R^F\}$$

▶ Example 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

11.3 Completeness Proofs (Optional Material, not Exam-Relevant)

11.3.1 Abstract Consistency and Model Existence

Model Existence (Overview)

- ▶ **Definition:** Abstract consistency
- ▶ **Definition:** Hintikka set (maximally abstract consistent)
- ▶ **Theorem:** Hintikka sets are satisfiable
- ▶ **Theorem:** If Φ is abstract consistent, then Φ can be extended to a Hintikka set.
- ▶ **Corollary:** If Φ is abstract consistent, then Φ is satisfiable
- ▶ **Application:** Let \mathcal{C} be a calculus, if Φ is \mathcal{C} -consistent, then Φ is abstract consistent.
- ▶ **Corollary:** \mathcal{C} is complete.

- ▶ Let \mathcal{C} be a calculus
- ▶ **Definition 3.1** Φ is called **\mathcal{C} -refutable**, if there is a formula \mathbf{B} , such that $\Phi \vdash_{\mathcal{C}} \mathbf{B}$ and $\Phi \vdash_{\mathcal{C}} \neg \mathbf{B}$.
- ▶ **Definition 3.2** We call a pair \mathbf{A} and $\neg \mathbf{A}$ a **contradiction**.
- ▶ So a set Φ is \mathcal{C} -refutable, if \mathcal{C} can derive a contradiction from it.
- ▶ **Definition 3.3** Φ is called **\mathcal{C} -consistent**, iff there is a formula \mathbf{B} , that is not derivable from Φ in \mathcal{C} .
- ▶ **Definition 3.4** We call a calculus \mathcal{C} **reasonable**, iff implication elimination and conjunction introduction are admissible in \mathcal{C} and $\mathbf{A} \wedge \neg \mathbf{A} \Rightarrow \mathbf{B}$ is a \mathcal{C} -theorem.
- ▶ **Theorem 3.5** \mathcal{C} -inconsistency and \mathcal{C} -refutability coincide for reasonable calculi.

Abstract Consistency

- ▶ **Definition 3.6** Let ∇ be a family of sets. We call ∇ **closed under subset** s, iff for each $\Phi \in \nabla$, all subsets $\Psi \subseteq \Phi$ are elements of ∇ .
- ▶ **Notation 3.7** We will use $\Phi * \mathbf{A}$ for $\Phi \cup \{\mathbf{A}\}$.
- ▶ **Definition 3.8** A family ∇ of sets of propositional formulae is called an **abstract consistency class**, iff it is closed under subsets, and for each $\Phi \in \nabla$
 - ∇_c) $P \notin \Phi$ or $\neg P \notin \Phi$ for $P \in \mathcal{V}_o$.
 - ∇_{\neg}) $\neg\neg \mathbf{A} \in \Phi$ implies $\Phi * \mathbf{A} \in \nabla$
 - ∇_{\vee}) $(\mathbf{A} \vee \mathbf{B}) \in \Phi$ implies $\Phi * \mathbf{A} \in \nabla$ or $\Phi * \mathbf{B} \in \nabla$
 - ∇_{\wedge}) $\neg(\mathbf{A} \vee \mathbf{B}) \in \Phi$ implies $(\Phi \cup \{\neg \mathbf{A}, \neg \mathbf{B}\}) \in \nabla$
- ▶ **Example 3.9** The empty set is an abstract consistency class
- ▶ **Example 3.10** The set $\{\emptyset, \{Q\}, \{P \vee Q\}, \{P \vee Q, Q\}\}$ is an abstract consistency class
- ▶ **Example 3.11** The family of satisfiable sets is an abstract consistency class.

Compact Collections

- ▶ **Definition 3.12** We call a collection ∇ of sets **compact**, iff for any set Φ we have
 $\Phi \in \nabla$, iff $\Psi \in \nabla$ for every finite subset Ψ of Φ .
- ▶ **Lemma 3.13** *If ∇ is compact, then ∇ is closed under subsets.*
- ▶ **Proof:**
 - P.1 Suppose $S \subseteq T$ and $T \in \nabla$.
 - P.2 Every finite subset A of S is a finite subset of T .
 - P.3 As ∇ is compact, we know that $A \in \nabla$.
 - P.4 Thus $S \in \nabla$. □

Compact Collections

- ▶ **Definition 3.14** We call a collection ∇ of sets **compact**, iff for any set Φ we have $\Phi \in \nabla$, iff $\Psi \in \nabla$ for every finite subset Ψ of Φ .
- ▶ **Lemma 3.15** *If ∇ is compact, then ∇ is closed under subsets.*
- ▶ **Proof:**
 - P.1 Suppose $S \subseteq T$ and $T \in \nabla$.
 - P.2 Every finite subset A of S is a finite subset of T .
 - P.3 As ∇ is compact, we know that $A \in \nabla$.
 - P.4 Thus $S \in \nabla$. □

Compact Abstract Consistency Classes I

► **Lemma 3.16** *Any abstract consistency class can be extended to a compact one.*

► **Proof:**

P.1 We choose $\nabla' := \{\Phi \subseteq wff_o(\mathcal{V}_o) \mid \text{every finite subset of } \Phi \text{ is in } \nabla\}$.

P.2 Now suppose that $\Phi \in \nabla$. ∇ is closed under subsets, so every finite subset of Φ is in ∇ and thus $\Phi \in \nabla'$. Hence $\nabla \subseteq \nabla'$.

P.3 Next let us show that each ∇' is compact.

P.3.1 Suppose $\Phi \in \nabla'$ and Ψ is an arbitrary finite subset of Φ .

P.3.2 By definition of ∇' all finite subsets of Φ are in ∇ and therefore $\Psi \in \nabla'$.

P.3.3 Thus all finite subsets of Φ are in ∇' whenever Φ is in ∇' .

P.3.4 On the other hand, suppose all finite subsets of Φ are in ∇' .

P.3.5 Then by the definition of ∇' the finite subsets of Φ are also in ∇ , so $\Phi \in \nabla$. Thus ∇' is compact.

P.4 Note that ∇' is closed under subsets by the Lemma above.

Compact Abstract Consistency Classes II

P.5 Now we show that if ∇ satisfies ∇_* , then ∇' satisfies ∇_* .

P.5.1 To show ∇_c , let $\Phi \in \nabla'$ and suppose there is an atom \mathbf{A} , such that $\{\mathbf{A}, \neg \mathbf{A}\} \subseteq \Phi$. Then $\{\mathbf{A}, \neg \mathbf{A}\} \in \nabla$ contradicting ∇_c .

P.5.2 To show ∇_{\neg} , let $\Phi \in \nabla'$ and $\neg \neg \mathbf{A} \in \Phi$, then $\Phi * \mathbf{A} \in \nabla'$.

P.5.2.1 Let Ψ be any finite subset of $\Phi * \mathbf{A}$, and $\Theta := (\Psi \setminus \{\mathbf{A}\}) * \neg \neg \mathbf{A}$.

P.5.2.2 Θ is a finite subset of Φ , so $\Theta \in \nabla$.

P.5.2.3 Since ∇ is an abstract consistency class and $\neg \neg \mathbf{A} \in \Theta$, we get $\Theta * \mathbf{A} \in \nabla$ by ∇_{\neg} .

P.5.2.4 We know that $\Psi \subseteq \Theta * \mathbf{A}$ and ∇ is closed under subsets, so $\Psi \in \nabla$.

P.5.2.5 Thus every finite subset Ψ of $\Phi * \mathbf{A}$ is in ∇ and therefore by definition $\Phi * \mathbf{A} \in \nabla'$.

P.5.3 the other cases are analogous to ∇_{\neg} . □

∇ -Hintikka Set I

- ▶ **Definition 3.17** Let ∇ be an abstract consistency class, then we call a set $\mathcal{H} \in \nabla$ a **∇ -Hintikka Set**, iff \mathcal{H} is maximal in ∇ , i.e. for all \mathbf{A} with $\mathcal{H} * \mathbf{A} \in \nabla$ we already have $\mathbf{A} \in \mathcal{H}$.
- ▶ **Theorem 3.18 (Hintikka Properties)** Let ∇ be an abstract consistency class and \mathcal{H} be a ∇ -Hintikka set, then
 - \mathcal{H}_c) For all $\mathbf{A} \in \text{wff}_o(\mathcal{V}_o)$ we have $\mathbf{A} \notin \mathcal{H}$ or $\neg \mathbf{A} \notin \mathcal{H}$
 - \mathcal{H}_{\neg}) If $\neg \neg \mathbf{A} \in \mathcal{H}$ then $\mathbf{A} \in \mathcal{H}$
 - \mathcal{H}_{\vee}) If $(\mathbf{A} \vee \mathbf{B}) \in \mathcal{H}$ then $\mathbf{A} \in \mathcal{H}$ or $\mathbf{B} \in \mathcal{H}$
 - \mathcal{H}_{\wedge}) If $\neg(\mathbf{A} \vee \mathbf{B}) \in \mathcal{H}$ then $\neg \mathbf{A}, \neg \mathbf{B} \in \mathcal{H}$

∇ -Hintikka Set II

► Proof:

P.1 We prove the properties in turn

P.1.1 \mathcal{H}_c : by induction on the structure of \mathbf{A}

P.1.1.1.1 $\mathbf{A} \in \mathcal{V}_o$: Then $\mathbf{A} \notin \mathcal{H}$ or $\neg \mathbf{A} \notin \mathcal{H}$ by ∇_c .

P.1.1.1.2 $\mathbf{A} = \neg \mathbf{B}$:

P.1.1.1.2.1 Let us assume that $\neg \mathbf{B} \in \mathcal{H}$ and $\neg \neg \mathbf{B} \in \mathcal{H}$,

P.1.1.1.2.2 then $\mathcal{H} * \mathbf{B} \in \nabla$ by ∇_{\neg} , and therefore $\mathbf{B} \in \mathcal{H}$ by maximality.

P.1.1.1.2.3 So both \mathbf{B} and $\neg \mathbf{B}$ are in \mathcal{H} , which contradicts the inductive hypothesis.

P.1.1.1.3 $\mathbf{A} = \mathbf{B} \vee \mathbf{C}$: similar to the previous case:

P.1.2 We prove \mathcal{H}_{\neg} by maximality of \mathcal{H} in ∇ :

P.1.2.1 If $\neg \neg \mathbf{A} \in \mathcal{H}$, then $\mathcal{H} * \mathbf{A} \in \nabla$ by ∇_{\neg} .

P.1.2.2 The maximality of \mathcal{H} now gives us that $\mathbf{A} \in \mathcal{H}$.

P.1.3 other \mathcal{H}_* are similar:

Extension Theorem

► **Theorem 3.19** If ∇ is an abstract consistency class and $\Phi \in \nabla$, then there is a ∇ -Hintikka set \mathcal{H} with $\Phi \subseteq \mathcal{H}$.

► **Proof:**

P.1 Wlog. we assume that ∇ is compact (otherwise pass to compact extension)

P.2 We choose an enumeration $\mathbf{A}^1, \mathbf{A}^2, \dots$ of the set $wff_o(\mathcal{V}_o)$

P.3 and construct a sequence of sets H^i with $H^0 := \Phi$ and

$$H^{n+1} := \begin{cases} H^n & \text{if } H^n * \mathbf{A}^n \notin \nabla \\ H^n * \mathbf{A}^n & \text{if } H^n * \mathbf{A}^n \in \nabla \end{cases}$$

P.4 Note that all $H^i \in \nabla$, choose $\mathcal{H} := \bigcup_{i \in \mathbb{N}} H^i$

P.5 $\Psi \subseteq \mathcal{H}$ finite implies there is a $j \in \mathbb{N}$ such that $\Psi \subseteq H^j$,

P.6 so $\Psi \in \nabla$ as ∇ closed under subsets and $\mathcal{H} \in \nabla$ as ∇ is compact.

P.7 Let $\mathcal{H} * \mathbf{B} \in \nabla$, then there is a $j \in \mathbb{N}$ with $\mathbf{B} = \mathbf{A}^j$, so that $\mathbf{B} \in H^{j+1}$ and $H^{j+1} \subseteq \mathcal{H}$

P.8 Thus \mathcal{H} is ∇ -maximal □

- ▶ **Definition 3.20** A function $\nu: wff_o(\mathcal{V}_o) \rightarrow \mathcal{D}_o$ is called a **valuation**, iff
 - ▶ $\nu(\neg \mathbf{A}) = \top$, iff $\nu(\mathbf{A}) = \text{F}$
 - ▶ $\nu(\mathbf{A} \vee \mathbf{B}) = \top$, iff $\nu(\mathbf{A}) = \top$ or $\nu(\mathbf{B}) = \top$
- ▶ **Lemma 3.21** If $\nu: wff_o(\mathcal{V}_o) \rightarrow \mathcal{D}_o$ is a valuation and $\Phi \subseteq wff_o(\mathcal{V}_o)$ with $\nu(\Phi) = \{\top\}$, then Φ is satisfiable.
- ▶ **Proof Sketch:** $\nu|_{\mathcal{V}_o}: \mathcal{V}_o \rightarrow \mathcal{D}_o$ is a satisfying variable assignment. □
- ▶ **Lemma 3.22** If $\varphi: \mathcal{V}_o \rightarrow \mathcal{D}_o$ is a variable assignment, then $\mathcal{I}_\varphi: wff_o(\mathcal{V}_o) \rightarrow \mathcal{D}_o$ is a valuation.

Model Existence

- ▶ **Lemma 3.23 (Hintikka-Lemma)** *If ∇ is an abstract consistency class and \mathcal{H} a ∇ -Hintikka set, then \mathcal{H} is satisfiable.*

- ▶ **Proof:**

- P.1 We define $\nu(\mathbf{A}) := \top$, iff $\mathbf{A} \in \mathcal{H}$

- P.2 then ν is a valuation by the Hintikka properties

- P.3 and thus $\nu|_{\mathcal{V}_0}$ is a satisfying assignment. □

- ▶ **Theorem 3.24 (Model Existence)** *If ∇ is an abstract consistency class and $\Phi \in \nabla$, then Φ is satisfiable.*

- ▶ **Proof:**

- ▶ P.1 There is a ∇ -Hintikka set \mathcal{H} with $\Phi \subseteq \mathcal{H}$ (Extension Theorem)
We know that \mathcal{H} is satisfiable. (Hintikka-Lemma)
In particular, $\Phi \subseteq \mathcal{H}$ is satisfiable. □

11.3.2 A Completeness Proof for Propositional Tableaux

Abstract Completeness for \mathcal{T}_0 I

P.8 Lemma 3.25 $\{\Phi \mid \Phi^T \text{ has no closed Tableau}\}$ is an abstract consistency class.

► **Proof:** Let's call the set above ∇

P.1 We have to convince ourselves of the abstract consistency properties

P.1.1 ∇_c : $P, \neg P \in \Phi$ implies $P^F, P^T \in \Phi^T$. □

P.1.2 ∇_{\neg} : Let $\neg\neg \mathbf{A} \in \Phi$.

P.1.2.1 For the proof of the contrapositive we assume that $\Phi * \mathbf{A}$ has a closed tableau \mathcal{T} and show that already Φ has one:

P.1.2.2 applying $\mathcal{T}_0\neg$ twice allows to extend any tableau with $\neg\neg \mathbf{B}^\alpha$ by \mathbf{B}^α .

P.1.2.3 any path in \mathcal{T} that is closed with $\neg\neg \mathbf{A}^\alpha$, can be closed by \mathbf{A}^α . □

Abstract Completeness for \mathcal{T}_0 II

P.1.3 ∇_V : Suppose $(\mathbf{A} \vee \mathbf{B}) \in \Phi$ and both $\Phi * \mathbf{A}$ and $\Phi * \mathbf{B}$ have closed tableaux

P.1.3.1 consider the tableaux:

$$\begin{array}{c} \Phi^T \\ \mathbf{A}^T \\ \text{Rest}^1 \end{array} \quad \begin{array}{c} \Phi^T \\ \mathbf{B}^T \\ \text{Rest}^2 \end{array} \quad \begin{array}{c} \Psi^T \\ \mathbf{A} \vee \mathbf{B}^T \\ \mathbf{A}^T \mid \mathbf{B}^T \\ \text{Rest}^1 \mid \text{Rest}^2 \end{array}$$

□

P.1.4 ∇_\wedge : suppose, $\neg(\mathbf{A} \vee \mathbf{B}) \in \Phi$ and $\Phi\{\neg\mathbf{A}, \neg\mathbf{B}\}$ have closed tableau \mathcal{T} .

P.1.4.1 We consider

$$\begin{array}{c} \Phi^T \\ \mathbf{A}^F \\ \mathbf{B}^F \\ \text{Rest} \end{array} \quad \begin{array}{c} \Psi^T \\ \mathbf{A} \vee \mathbf{B}^F \\ \mathbf{A}^F \\ \mathbf{B}^F \\ \text{Rest} \end{array}$$

where $\Phi = \Psi * \neg(\mathbf{A} \vee \mathbf{B})$.

□

□

Completeness of \mathcal{T}_0

► **Corollary 3.26** \mathcal{T}_0 is complete.

► **Proof:** by contradiction

P.1 We assume that $\mathbf{A} \in wff_o(\mathcal{V}_o)$ is valid, but there is no closed tableau for \mathbf{A}^F .

P.2 We have $\{\neg \mathbf{A}\} \in \nabla$ as $\neg \mathbf{A}^T = \mathbf{A}^F$.

P.3 so $\neg \mathbf{A}$ is satisfiable by the model existence theorem (which is applicable as ∇ is an abstract consistency class by our Lemma above)

P.4 this contradicts our assumption that \mathbf{A} is valid. □

Part III Legal Foundations of Information Technology

Chapter 12 Intellectual Property, Copyright, and Licensing

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 0.1** 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.

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

- ▶ The various legal systems of the world can be grouped into “traditions”.
- ▶ **Definition 0.2** Legal systems in the **common law tradition** are usually based on case law, they are often derived from the British system.
- ▶ **Definition 0.3** 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.

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 0.4** 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 0.5** 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

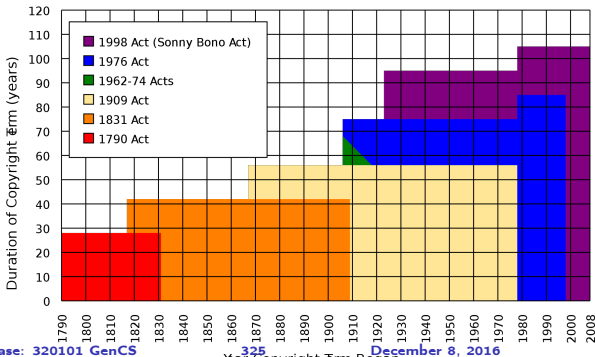
12.1 Copyright

Copyrightable Works

- ▶ **Definition 1.1** A **copyrightable work** is any artefact of human labor that fits into one of the following eight categories:
 - ▶ **Literary works**: 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 recordings** 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)

Limitations of Copyrightability: The Public Domain

- ▶ **Definition 1.2** A work is said to be in the **public domain**, if no copyright applies, otherwise it is called **copyrighted**.
- ▶ **Example 1.3** Works made by US government employees (in their work time) are in the public domain directly (**Rationale: taxpayer already payed for them**)
- ▶ **Copyright expires**: usually 70 years after the death of the creator
- ▶ **Example 1.4 (US Copyright Terms)** Some people claim that US copyright terms are extended, whenever Disney's Mickey Mouse would become public domain.



- ▶ **Definition 1.5** 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 1.6 (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 hires the employer, in jurisdictions from the civil law tradition, the author, unless the respective contract regulates it otherwise.*

Rights under Copyright Law

- ▶ **Definition 1.7** 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 1.8** 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.

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 1.9 (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.

12.2 Licensing

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 2.1** 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.

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 2.2** **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.

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 2.3** A **copyleft** license is a license which requires that allows derivative works, but requires that they be licensed with the same license.
- ▶ **Definition 2.4** 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)
- ▶ Various software companies have developed successful business models based on FLOSS licensing models. (e.g. Red Hat, Mozilla, IBM, ...)

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 2.5** The **Creative Commons licenses** 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)



Chapter 13 Information Privacy

Information/Data Privacy

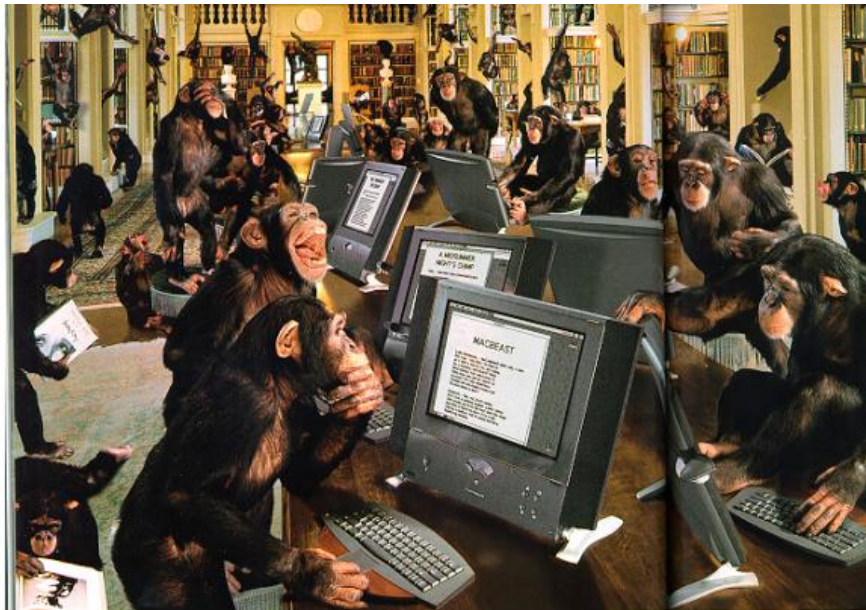
- ▶ **Definition 0.1** The principle of **information privacy** comprises the idea that humans have the right to control who can access their personal data when.
- ▶ Information privacy concerns exist wherever personally identifiable information is collected and stored – in digital form or otherwise. 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.

Part IV A look back; What have we learned?

Let's hack!  \rightsquigarrow 2am in the CLAMV cluster



⚠ no, let's think ⚠

- ▶ GIGO: Garbage In, Garbage Out (– ca. 1967)
- ▶ Applets, Not Craplets™ (– ca. 1997)

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!)

Machine Models

- ▶ Abstract Interpreters (mind games)
- ▶ The SML interpreter/compiler (didn't we love recursive programming?)

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 (\mathcal{O} oooooh, how fast this class grows)






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)



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

References I

-  Gerhard Gentzen.
Untersuchungen über das logische Schließen I.
Mathematische Zeitschrift, 39:176–210, 1935.
-  Paul R. Halmos.
Naive Set Theory.
Springer Verlag, 1974.
-  Harry R. Lewis and Christos H. Papadimitriou.
Elements of the Theory of Computation.
Prentice Hall, 1998.
-  Neil/Fred's gigantic list of palindromes.
web page at <http://www.derf.net/palindromes/>.
-  Lawrence C. Paulson.
ML for the working programmer.
Cambridge University Press, 1991.

References II

-  Kenneth H. Rosen.
Discrete Mathematics and Its Applications.
McGraw-Hill, 1990.
-  The Standard ML basis library, 2010.