Symbolic Methods for AI

Prof. Dr. Michael Kohlhase

Knowledge Representation and -Processing
Computer Science, FAU Erlangen-Nürnberg
Michael.Kohlhase@FAU.de

2025-05-06

Elevator Pitch for Symbolic Methods for AI

- **Mission:** In this course we will try to give students all the prerequisites (theoretical and practical) for surviving courses in symbolic artificial intelligence.

- **There is no shame in needing SMAI:** Not all undergraduate programs value these topics and focus on them. (The Master AI at FAU does!)

Elevator Pitch for Symbolic Methods for AI

- **Mission:** In this course we will try to give students all the prerequisites (theoretical and practical) for surviving courses in symbolic artificial intelligence.
- **You will not need this course if you . . .**
  - . . . have practical experience in programming
  - . . . and understand and can converse effectively in
    1. discrete mathematics, e.g. sets, functions, relations, products, power sets, quotients, trees, graphs, . . .
    2. transition systems, automata, Turing machines,
    3. context-free grammars, languages, syntax trees,
    4. mathematical structures, in particular the magma and bin-rel hierarchies,
  - . . . and understand and can apply
    1. mathematical argumentation and proofs, in particular all forms of induction,
    2. computational complexity (the underlying concepts) and can diagnose the complexity classes of problems and algorithms,
    3. symbolic programming (i.e. recursive functions, (syntax) tree traversal, option, list, record datatypes . . .)
    4. typed programming languages, recursive programming (functional/logic programming)
    5. formal proof systems (propositional logic).
- **There is no shame in needing SMAI:** Not all undergraduate programs value these topics and focus on them. (The Master AI at FAU does!)

Chapter 1
Preliminaries

1.1 Administrative Ground Rules

Prerequisites for SMAI

- **Content Prerequisites:** The equivalent to a bachelor in CS (or CS+X) outside of FAU.
- **Background:** The formal prerequisite of the Master Artificial Intelligence is "a bachelor degree equivalent to a CS B.Sc from FAU".

Prerequisites for SMAI

- **Content Prerequisites:** The equivalent to a bachelor in CS (or CS+X) outside of FAU.
- **Background:** The formal prerequisite of the Master Artificial Intelligence is "a bachelor degree equivalent to a CS B.Sc from FAU".
- **Intuition:** SMAI is a remedial course if you do not have an education that is!

Prerequisites for SMAI

- **Content Prerequisites:** The equivalent to a bachelor in CS (or CS+X) outside of FAU.
- **Background:** The formal prerequisite of the Master Artificial Intelligence is "a bachelor degree equivalent to a CS B.Sc from FAU".
- **Intuition:** SMAI is a remedial course if you do not have an education that is!
- **The real Prerequisite:** Motivation, interest, curiosity, hard work. (SMAI is non-trivial)
- You ckan do this course if you want! (We will help you)

Assessment, Grades

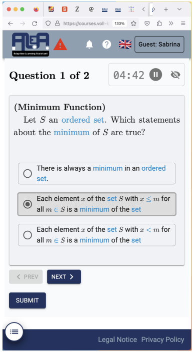
- **Overall (Module) Grade:**
  - Grade via the exam (Klausur) $\leadsto$ 100% of the grade.
  - Up to 10% bonus on-top for an exam with $\geq$ 50% points. ($<$ 50% $\leadsto$ no bonus)
  - Bonus points $\widehat{=}$ percentage sum of the best 10 prepquizzes divided by 100.

Assessment, Grades

- **Overall (Module) Grade:**
  - Grade via the exam (Klausur) $\leadsto$ 100% of the grade.
  - Up to 10% bonus on-top for an exam with $\geq 50\%$ points. ($< 50\% \leadsto$ no bonus)
  - Bonus points $\widehat{=}$ percentage sum of the best 10 prepquizzes divided by 100.
- **Exam:** exam conducted in presence on paper! ($\sim$ October 8. 2025)
- **Retake Exam:** 60 minutes exam six months later. ($\sim$ April. 8. 2026)
- ⚠ You have to register for exams in `https://campo.fau.de` in the first month of classes.
- **Note:** You can de-register from an exam on `https://campo.fau.de` up to three working days before exam. (do not miss that if you are not prepared)

Preparedness Quizzes

- **PrepQuizzes:** Before every lecture we offer a 10 min online quiz – the PrepQuiz – about the material from the previous week. ($\sim$ 10:0?-10:15 (check on ALEA); starts in week 2)
- **Motivations:** We do this to
  - keep you prepared and working continuously. (primary)
  - bonus points if the exam has $\geq 50\%$ points (potential part of your grade)
  - update the ALEA learner model. (fringe benefit)
- The prepquizes will be given in the ALEA system



- `https://courses.voll-ki.fau.de/quiz-dash/smai`
- You have to be logged into ALEA! (via FAU IDM)
- You can take the prepquiz on your laptop or phone, ...
- ... in the lecture or at home ...
- ... via WLAN or 4G Network. (do not overload)
- Prepquizzes will only be available $\sim$ 10:0?-10:15 (check on ALEA)!

Next Week: Pretest

- ⚠ Next week we will try out the prepquiz infrastructure with a pretest!
  - **Presence**: bring your laptop or cellphone.
  - **Online**: you can and should take the pretest as well.
  - Have a recent firefox or chrome (chrome: younger than March 2023)
  - Make sure that you are logged into ALeA (via FAU IDM; see below)
- **Definition 1.1.** A pretest is an assessment for evaluating the preparedness of learners for further studies.
- **Concretely:** This pretest
  - establishes a baseline for the competency expectations in and
  - tests the ALeA quiz infrastructure for the prepquizzes.
- Participation in the pretest is optional; it will not influence grades in any way.
- The pretest covers the prerequisites of SMAI and some of the material that may have been covered in other courses.
- The test will be also used to refine the ALeA learner model, which may make learning experience in ALeA better. (see below)

Take SMAI Seriously!

- The course SMAI was intended as a prep-course for symbolic AI. (concretely AI-1).
- So really, it is intended for first-semester Master AI students.
- **Observation:** Over 3/4 of you are higher-semester student (HSS) in the Master AI. (and I completely understand why you are taking it ⇜ 2.5 ECTS ☺)
- **Non-/Consequences:** I will still teach the course as a prep-course for AI-1
  - HSS should consider themselves as tolerated guests (not the focus)
  - SMAI will cover many that help with symbolic AI, but were not taught in AI-1
  - SMAI will teach things explicitly that you would otherwise have learned by osmosis
  - Even if you passed AI-1, you will not pass SMAI without real work.
- ⚠ Take SMAI seriously! (otherwise you may have difficulties passing)
  - be present in the lectures (positive correlation with learning/passing)
  - do the homework problems, and also participate in peer grading (ditto)
  - take all the quizzes, look at and try to understand the results (learn from them)
  - form a study group to discuss the course contents critically.

1.2 Getting Most out of SMAI

SMAI Homework Assignments

- **Goal:** Homework assignments reinforce what was taught in lectures.
- **Homework Assignments:** Small individual problem/programming/proof task
  - but take time to solve (at least read them directly $\leadsto$ questions)
- **Didactic Intuition:** Homework assignments give you material to test your understanding and show you how to apply it.
- ⚠ Homeworks give no points, but without trying you are unlikely to pass the exam.
- **Our Experience:** Doing your homework is probably even *more* important (and predictive of exam success) than attending the lecture in person!

SMAI Homework Assignments

- **Goal:** Homework assignments reinforce what was taught in lectures.
- **Homework Assignments:** Small individual problem/programming/proof task
  - but take time to solve (at least read them directly $\leadsto$ questions)
- **Didactic Intuition:** Homework assignments give you material to test your understanding and show you how to apply it.
- Homeworks give no points, but without trying you are unlikely to pass the exam.
- **Our Experience:** Doing your homework is probably even *more* important (and predictive of exam success) than attending the lecture in person!
- Homeworks will be mainly peer-graded in the ALeA system.
- **Didactic Motivation:** Through peer grading students are able to see mistakes in their thinking and can correct any problems in future assignments. By grading assignments, students may learn how to complete assignments more accurately and how to improve their future results. (not just us being lazy)

SMAI Homework Assignments – Howto

- **Homework Workflow:** in ALeA (see below)
  - Homework assignments will be published on thursdays: see `https://courses.voll-ki.fau.de/hw/smai`
  - Submission of solutions via the ALeA system in the week after
  - Peer grading/feedback (and master solutions) via answer classes.
- **Quality Control:** TAs and instructors will monitor and supervise peer grading.

SMAI Homework Assignments – Howto

- **Homework Workflow:** in ALeA (see below)
  - Homework assignments will be published on thursdays: see `https://courses.voll-ki.fau.de/hw/smai`
  - Submission of solutions via the ALeA system in the week after
  - Peer grading/feedback (and master solutions) via answer classes.
- **Quality Control:** TAs and instructors will monitor and supervise peer grading.
- **Experiment:** Can we motivate enough of you to make peer assessment self-sustaining?
  - I am appealing to your sense of community responsibility here ...
  - You should only expect other's to grade your submission if you grade their's (cf. Kant's "Moral Imperative")
  - **Make no mistake**: The grader usually learns at least as much as the gradee.

SMAI Homework Assignments – Howto

- **Homework Workflow:** in ALeA (see below)
  - Homework assignments will be published on thursdays: see `https://courses.voll-ki.fau.de/hw/smai`
  - Submission of solutions via the ALeA system in the week after
  - Peer grading/feedback (and master solutions) via answer classes.
- **Quality Control:** TAs and instructors will monitor and supervise peer grading.
- **Experiment:** Can we motivate enough of you to make peer assessment self-sustaining?
  - I am appealing to your sense of community responsibility here . . .
  - You should only expect other's to grade your submission if you grade their's (cf. Kant's "Moral Imperative")
  - **Make no mistake**: The grader usually learns at least as much as the gradee.
- **Homework/Tutorial Discipline:**
  - Start early! (many assignments need more than one evening's work)
  - Don't start by sitting at a blank screen (talking & study groups help)
  - Humans will be trying to understand the text/code/math when grading it.
  - Go to the tutorials, discuss with your TA! (they are there for you!)

Collaboration

- **Definition 2.1.** Collaboration (or cooperation) is the process of groups of agents acting together for common, mutual benefit, as opposed to acting in competition for selfish benefit. In a collaboration, every agent contributes to the common goal and benefits from the contributions of others.
- In learning situations, the benefit is "better learning".
- **Observation:** In collaborative learning, the overall result can be significantly better than in competitive learning.
- **Good Practice:** Form study groups. (long- or short-term)
  1. ⚠ Those learners who work/help most, learn most!
  2. ⚠ Freeloaders – individuals who only watch – learn very little!
- It is OK to collaborate on homework assignments in SMAI! (no bonus points)
- Choose your study group well! (ALeA helps via the study buddy feature)

Do I need to attend the SMAI Lectures

- Attendance is not mandatory for the SMAI course. (official version)
- **Note:** There are two ways of learning: (both are OK, your mileage may vary)
  - Approach B: Read a book/papers (here: lecture notes)
  - Approach I: come to the lectures, be involved, interrupt the instructor whenever you have a question.

  The only advantage of I over B is that books/papers do not answer questions
- Approach S: come to the lectures and sleep does not work!
- The closer you get to research, the more we need to discuss!

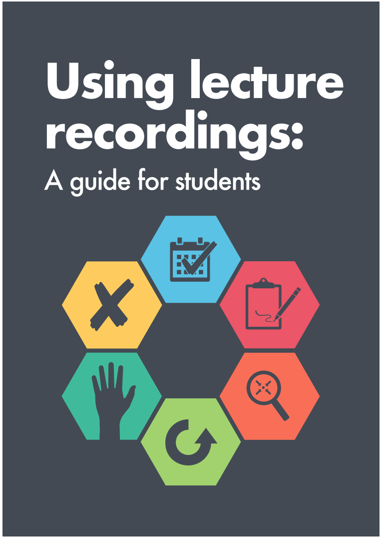
1.3 Learning Resources for SMAI

Course Notes, Forum, Matrix

- **Lecture notes** will be posted at `https://kwarc.info/teaching/SMAI`
  - We mostly prepare/update them as we go along (semantically preloaded ⇝ research resource)
  - Please report any errors/shortcomings you notice. (improve for the group/successors)
- **StudOn Forum:** For announcements – `https://www.studon.fau.de/studon/goto.php?target=lcode_uuXSYH8s`
- **Matrix Channel:** `https://matrix.to/#/#smai:fau.de` for questions, discussion with instructors and among your fellow students. (your channel, use it!)
  Login via FAU IDM ⇝ instructions
- **Course Videos** are at at `https://www.fau.tv/course/id/4226`.
- **Do not let the videos mislead you:** Coming to class is highly correlated with passing the exam!

Practical recommendations on Lecture Videos

- **Excellent Guide:** [Nor+18a] (German version at [Nor+18b])

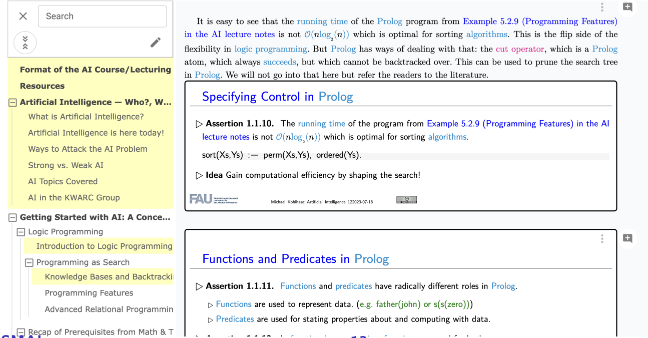


- Attend lectures.
- Take notes.
- Be specific.
- Catch up.
- Ask for help.
- Don't cut corners.

1.3.1 ALeA – AI-Supported Learning

ALEA: Adaptive Learning Assistant

- **Idea:** Use AI methods to help teach/learn AI (AI4AI)
- **Concretely:** Provide HTML versions of the SMAI slides/lecture notes and embed learning support services into them. (for pre/postparation of lectures)
- **Definition 3.1.** Call a document active, iff it is interactive and adapts to specific information needs of the readers. (lecture notes on steroids)
- **Intuition:** ALEA serves active course materials. (PDF mostly inactive)
- **Goal:** Make ALEA more like a instructor + study group than like a book!
- **Example 3.2 (Course Notes).** $\widehat{=}$ Slides + Comments



VoLL-KI Portal at https://courses.voll-ki.fau.de

- **Portal for ALeA Courses:** https://courses.voll-ki.fau.de

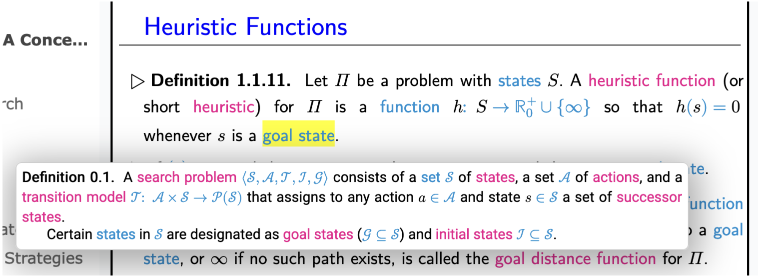
- **SMAI in ALeA:** https://courses.voll-ki.fau.de/course-home/smai
  - All details for the course.
  - recorded syllabus (keep track of material covered in course)
  - syllabus of the last semesters (for over/preview)
- **ALeA Status:** The ALEA system is deployed at FAU for over 1000 students taking eight courses
  - (some) students use the system actively (our logs tell us)
  - reviews are mostly positive/enthusiastic (error reports pour in)

Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.

Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 3.6 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)



A Conce...

rch

at

Strategies

Heuristic Functions

▷ **Definition 1.1.11.** Let Π be a problem with states S. A heuristic function (or short heuristic) for Π is a function $h\colon S \to \mathbb{R}^+_0 \cup \{\infty\}$ so that $h(s) = 0$ whenever s is a goal state.

...te.

...unction

... a goal

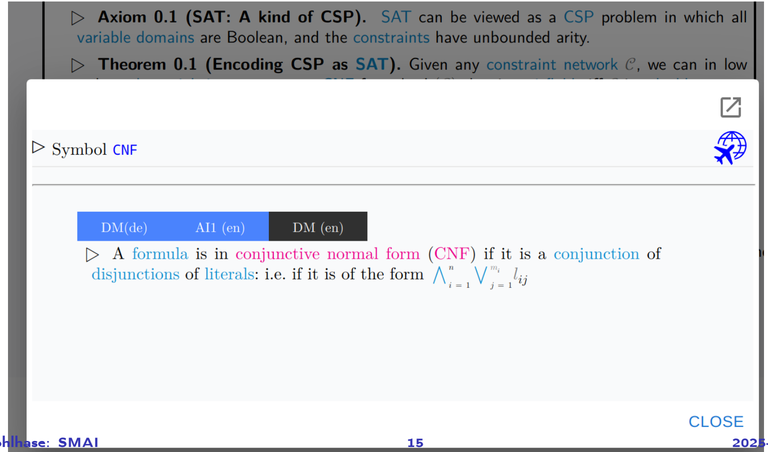
state, or ∞ if no such path exists, is called the goal distance function for Π.

Definition 0.1. A search problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set $\mathcal{S}$ of states, a set $\mathcal{A}$ of actions, and a transition model $\mathcal{T}\colon \mathcal{A} \times \mathcal{S} \to \mathcal{P}(\mathcal{S})$ that assigns to any action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$ a set of successor states.

Certain states in $\mathcal{S}$ are designated as goal states ($\mathcal{G} \subseteq \mathcal{S}$) and initial states $\mathcal{I} \subseteq \mathcal{S}$.

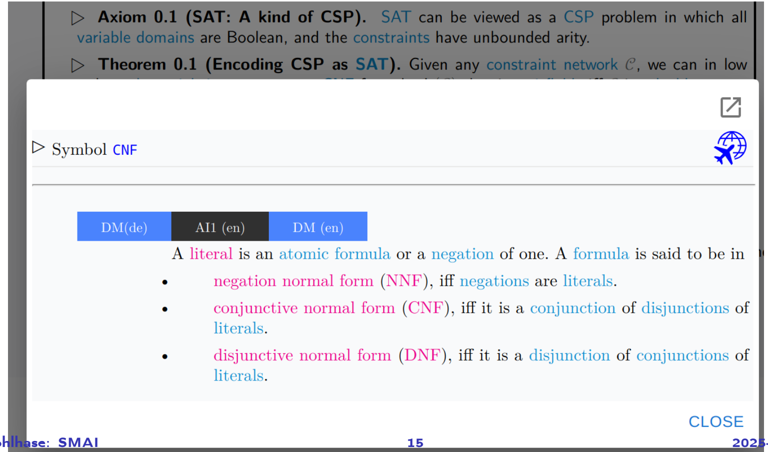
Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 3.9 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)
- **Example 3.10 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.



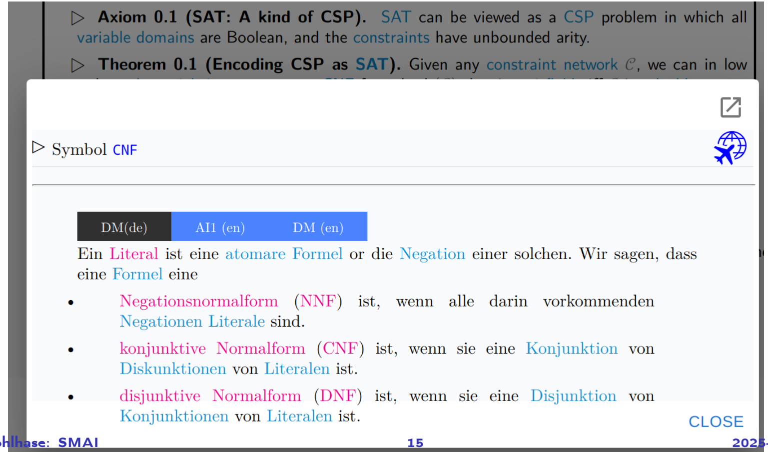
Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 3.12 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)
- **Example 3.13 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.



Learning Support Services in ALeA

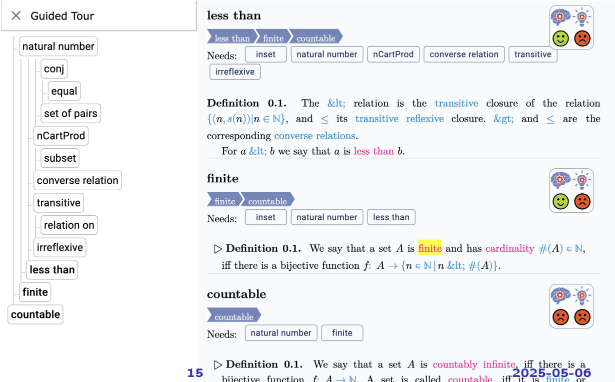
- **Idea:** Embed learning support services into active course materials.
- **Example 3.15 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)
- **Example 3.16 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.



Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 3.18 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)
- **Example 3.19 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.
- **Example 3.20 (Guided Tour).** A guided tour for a concept c assembles definitions/etc. into a self-contained mini-course culminating at c.

$c =$ countable $\rightsquigarrow$



Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 3.21 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)
- **Example 3.22 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.
- **Example 3.23 (Guided Tour).** A guided tour for a concept c assembles definitions/etc. into a self-contained mini-course culminating at c.
- ... your idea here ... (the sky is the limit)

(Practice/Remedial) Problems Everywhere

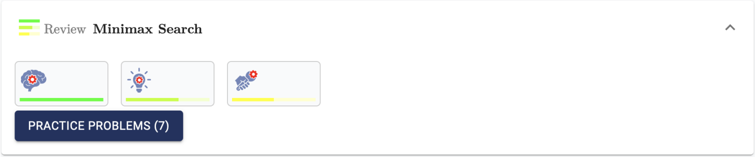
- **Problem:** Learning requires a mix of understanding and test-driven practice.
- **Idea:** ALeA supplies targeted practice problems everywhere.
- **Concretely:** Revision markers at the end of sections.

(Practice/Remedial) Problems Everywhere

- **Problem:** Learning requires a mix of understanding and test-driven practice.
- **Idea:** ALeA supplies targeted practice problems everywhere.
- **Concretely:** Revision markers at the end of sections.
  - A relatively non-intrusive overview over competency

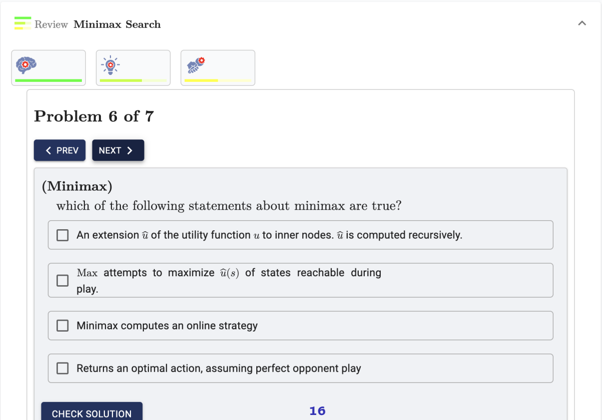
(Practice/Remedial) Problems Everywhere

- **Problem:** Learning requires a mix of understanding and test-driven practice.
- **Idea:** ALeA supplies targeted practice problems everywhere.
- **Concretely:** Revision markers at the end of sections.
  - A relatively non-intrusive overview over competency
  - Click to extend it for details.



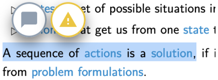
(Practice/Remedial) Problems Everywhere

- **Problem:** Learning requires a mix of understanding and test-driven practice.
- **Idea:** ALeA supplies targeted practice problems everywhere.
- **Concretely:** Revision markers at the end of sections.
  - A relatively non-intrusive overview over competency
  - Click to extend it for details.
  - Practice problems as usual. (targeted to your specific competency)



Localized Interactions with the Community

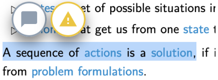
- Selecting text brings up localized – i.e. anchored on the selection – interactions:



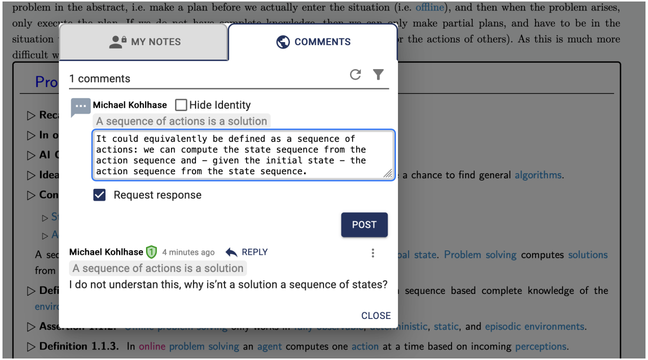
  - post a (public) comment or take (private) note
  - report an error to the course authors/instructors

Localized Interactions with the Community

- Selecting text brings up localized – i.e. anchored on the selection – interactions:



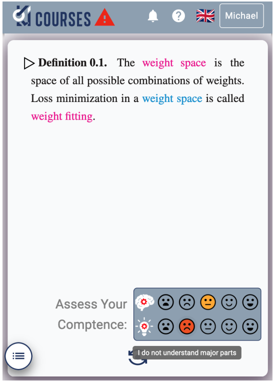
  - post a (public) comment or take (private) note
  - report an error to the course authors/instructors
- Localized comments induce a thread in the ALEA forum (like the StudOn Forum, but targeted towards specific learning objects.)



- Answering questions gives karma $\widehat{=}$ a public measure of user helpfulness.
- Notes can be anonymous (... generate no karma)

New Feature: Drilling with Flashcards

- Flashcards challenge you with a task (term/problem) on the front...



  ...and the definition/answer is on the back.
- Self-assessment updates the learner model (before/after)
- **Idea:** Challenge yourself to a card stack, keep drilling/assessing flashcards until the learner model eliminates all.
- **Bonus:** Flashcards can be generated from existing semantic markup (educational equivalent to free beer)

Learner Data and Privacy in ALeA

- **Observation:** Learning support services in ALeA use the learner model; they
  - need the learner model data to adapt to the invidivual learner!
  - collect learner interaction data (to update the learner model)
- **Consequence:** You need to be logged in (via your FAU IDM credentials) for useful learning support services!

Learner Data and Privacy in ALeA

- **Observation:** Learning support services in ALeA use the learner model; they
  - need the learner model data to adapt to the invidivual learner!
  - collect learner interaction data (to update the learner model)
- **Consequence:** You need to be logged in (via your FAU IDM credentials) for useful learning support services!
- **Problem:** Learner model data is highly sensitive personal data!
- **ALeA Promise:** The ALeA team does the utmost to keep your personal data safe. (SSO via FAU IDM/eduGAIN, ALeA trust zone)

Learner Data and Privacy in ALeA

- **Observation:** Learning support services in ALeA use the learner model; they
  - need the learner model data to adapt to the invidivual learner!
  - collect learner interaction data (to update the learner model)
- **Consequence:** You need to be logged in (via your FAU IDM credentials) for useful learning support services!
- **Problem:** Learner model data is highly sensitive personal data!
- **ALeA Promise:** The ALeA team does the utmost to keep your personal data safe. (SSO via FAU IDM/eduGAIN, ALeA trust zone)
- **ALeA Privacy Axioms:**
  1. ALeA only collects learner models data about logged in users.
  2. Personally identifiable learner model data is only accessible to its subject (delegation possible)
  3. Learners can always query the learner model about its data.
  4. All learner model data can be purged without negative consequences (except usability deterioration)
  5. Logging into ALeA is completely optional.
- **Observation:** Authentication for bonus quizzes are somewhat less optional, but you can always purge the learner model later.

Concrete Todos for ALeA

- **Recall:** You will use ALeA for the prepquizzes (or lose bonus points)
  All other use is optional. (but AI-supported pre/postparation can be helpful)
- To use the ALeA system, you will have to log in via SSO: (do it now)
  - go to `https://courses.voll-ki.fau.de/course-home/smai`,
  - in the upper right hand corner you see 

,
  - log in via your FAU IDM credentials. (you should have them by now)
  - You get access to your personal ALeA profile via 

(plus feature notifications, manual, and language chooser)

Concrete Todos for ALeA

- **Recall:** You will use ALeA for the prepquizzes (or lose bonus points)
  All other use is optional. (but AI-supported pre/postparation can be helpful)
- To use the ALeA system, you will have to log in via SSO: (do it now)
  - go to `https://courses.voll-ki.fau.de/course-home/smai`,
  - in the upper right hand corner you see 

,
  - log in via your FAU IDM credentials. (you should have them by now)
  - You get access to your personal ALeA profile via 

    (plus feature notifications, manual, and language chooser)
- **Problem:** Most ALeA services depend on the learner model. (to adapt to you)
- **Solution:** Initialize your learner model with your educational history!
  - **Concretely**: enter taken CS courses (FAU equivalents) and grades.
  - ALeA uses that to estimate your CS/AI competencies. (for your benefit)
  - then ALeA knows about you; I don't! (ALeA trust zone)

Chapter 2
Foundations: Mathematical Language in Practice

Let's start with the math!

Discrete Math for the moment

- Kenneth H. Rosen *Discrete Mathematics and Its Applications* [Ros90].
- Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation* [LP98].
- Paul R. Halmos, *Naive Set Theory* [Hal74].

2.1 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,
- **Definition 1.7.** 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 1.8.** For ////, apply the *o*-rule once and then the *s*-rule four times.
- **Definition 1.9.** we call these representations unary natural numbers.

A little more sophistication (math) please

- **Definition 1.10.** We call a unary natural number the successor (predecessor) of another, if it can be constructing by adding (removing) a slash. (successors are created by the *s*-rule)
- **Example 1.11.** /// is the successor of // and // the predecessor of ///.
- **Definition 1.12.** The following set of axioms are called the Peano axioms (Giuseppe Peano ∗1858, †1932)
- **Axiom 1.13 (P1).** *" " (aka. "zero") is a unary natural number. " " (aka. "zero") is a unary natural number.*
- **Axiom 1.14 (P2).** *Every unary natural number has a successor that is a unary natural number and that is different from it.*
- **Axiom 1.15 (P3).** *Zero is not a successor of any unary natural number.*
- **Axiom 1.16 (P4).** *Different unary natural numbers have different successors.*
- **Axiom 1.17 (P5: Induction Axiom).** *Every unary natural number possesses a property P, if*
  - *zero has property P and* (*base case*)
  - *the successor of every unary natural number that has property P also possesses property P.* (*step case*)
- **Question:** Why is this a better way of saying things (why so complicated?)

2.2 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:

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

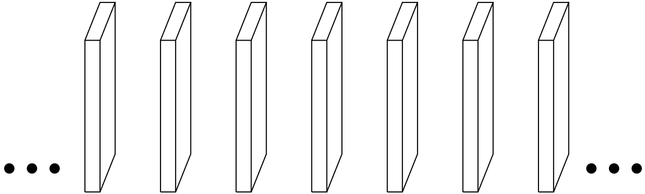
  1. ' ' is zero, so ' ' is "zero or a successor".
  2. Let n be a arbitrary unary natural number that "is zero or a successor"
  3. Then its successor "is a successor", so the successor of n is "zero or a successor"
  4. Since we have taken n arbitrary (nothing in our argument depends on the choice) we have shown that for any n, its successor has property P.
  5. Property P holds for all unary natural numbers by [method=apply]P5, so we have proven the assertion

  □

The Domino Theorem

- **Theorem 2.11.** *Let $S_1, S_2, \ldots$ 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 $S_{s(i)}$".
  We have to consider two cases
  1. base case: i is zero
     1.1. We have assumed that "S_0 is pushed towards S_1, so that it falls"
  3. step case: $i = s(j)$ for some unary natural number j
     3.1. We assume that P holds for S_j, i.e. S_j falls in the direction of $S_{s(j)} = S_i$.
     3.2. But we know that S_j has the same weight as S_i, which is unstable,
     3.3. so S_i falls into the direction opposite to S_j, i.e. towards $S_{s(i)}$ (we have a linear sequence of dominos)
  5. We have considered all the cases, so we have proven that P holds for all unary natural numbers i. (by induction)
  6. Now, the assertion follows trivially, since if "S_i falls in the direction of $S_{s(i)}$", then in particular "S_i falls".

$\square$

2.3 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: $n \oplus o = n$
  - adding m to the successor of n yields the successor of $m \oplus n$.
    written as an equation: $m \oplus s(n) = s(m \oplus n)$
- **Questions:** to understand this definition, we have to know
  - Is this "definition" well-formed? (does it characterize a mathematical object?)
  - May we define "functions" by algorithms? (what is a function anyways?)

Addition on unary natural numbers is associative

- **Theorem 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
  1. The property of l is that $n \oplus m \oplus l = n \oplus m \oplus l$ holds.
  2. We have to consider two cases
     2.1. base case
        2.1.1. $n \oplus m \oplus o = n \oplus m = n \oplus m \oplus o$
     2.3. step case
        2.3.1. given arbitrary l, assume $n \oplus m \oplus l = n \oplus m \oplus l$, show $n \oplus m \oplus s(l) = n \oplus m \oplus s(l)$.
        2.3.2. We have $n \oplus m \oplus s(l) = n \oplus s(m \oplus l) = s(n \oplus m \oplus l)$
        2.3.3. By induction hypothesis $s(n \oplus m \oplus l) = n \oplus m \oplus s(l)$

$\square$

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 $\bigodot_{i=o}^{s(m)}(n_i) = n_{s(m)} \odot \bigodot_{i=o}^{m}(n_i)$.

Chapter 3
Talking (and Writing) about Mathematics

Talking about Mathematics

- **Definition 0.1.** Mathematicians use a stylized language that
  - uses formulae to represent mathematical objects, e.g. $\int_1^0 x^{3/2} dx$
  - 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 0.2.** A technical language is a natural language extended by a terminology and (possibly) special idioms, discourse markers, and notations.
- **Definition 0.3.** A jargon (or terminology) is a set of specialized words or phrases (called technical terms or just terms) relating to concepts from a particular domain of discourse.
- **Observation:** Mathematical vernacular is a technical language that you need to master to be successful when moving to a new environment – symbolic AI.
- Like you should learn German when moving to Germany (to buy bread in the local bakery)

3.1 Talking about Mathematical Objects

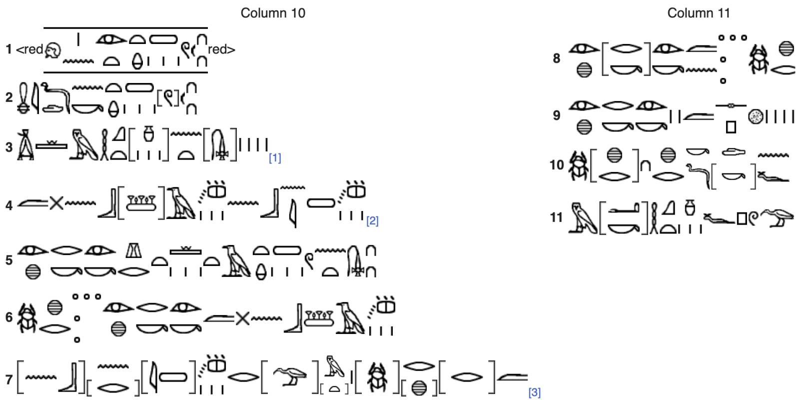
Excusion: Math Problems in Antiquity

- **Example 1.1 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
    - A pefsu measures the strength of the beer made from a heqat of grain.
    - A higher pefsu number means weaker bread or beer.
    - The unit pefsu appears in many offering lists (that decorate the outer walls of temples)

Excusion: Math Problems in Antiquity

- **Example 1.3 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
  - The hieroglypic transliteration of Problem 8: (about pefsu)

Problem 8 [edit]



Excusion: Math Problems in Antiquity

- **Example 1.5 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
  - The hieroglypic transliteration of Problem 8: (about pefsu)
  - If you do not read hieroglypics:
    1. *Example of calculating 100 loaves of bread of pefsu 20*
    2. *If someone says to you: "You have 100 loaves of bread of pefsu 20 to be exchanged for beer of pefsu 4 like 1/2 1/4 malt-date beer"*
    3. *First calculate the grain required for the 100 loaves of the bread of pefsu 20*
    4. *The result is 5 heqat. Then reckon what you need for a des-jug of beer like the beer called 1/2 1/4 malt-date beer*
    5. *The result is 1/2 of the heqat measure needed for des-jug of beer made from upper-Egyptian grain.*
    6. *Calculate 1/2 of 5 heqat, the result will be 21/2*
    7. *Take this 21/2 four times*
    8. *The result is 10. Then you say to him:*
    9. *"Behold! The beer quantity is found to be correct".*

Excusion: Math Problems in Antiquity

- **Example 1.7 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
  - The hieroglypic transliteration of Problem 8: (about pefsu)
  - If you do not read hieroglypics:
  - We would specify this today as (much more efficient!)

$$\text{pefsu} = \frac{\text{number loaves of bread (or jugs of beer)}}{\text{number of heqats of grain}}$$

Excusion: Math Problems in Antiquity

- **Example 1.9 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
  - The hieroglypic transliteration of Problem 8: (about pefsu)
  - If you do not read hieroglypics:
  - We would specify this today as (much more efficient!)

$$\text{pefsu} = \frac{\text{number loaves of bread (or jugs of beer)}}{\text{number of heqats of grain}}$$

  - Even better: The modern notation comes with extablished calculation rules! (remember school?)

Excusion: Math Problems in Antiquity

- **Example 1.11 (In Egypt).** Problem 8 from the Moscow Mathematical Papyrus
  - Background: pefsu as unit of measurement
  - The hieroglypic transliteration of Problem 8: (about pefsu)
  - If you do not read hieroglypics:
  - We would specify this today as (much more efficient!)

$$\text{pefsu} = \frac{\text{number loaves of bread (or jugs of beer)}}{\text{number of heqats of grain}}$$

  - Even better: The modern notation comes with extablished calculation rules! (remember school?)
- **Example 1.12 (In Ancient Rome).** Some of the highest-payed specialists in Caesar's campaign in Gallia were "computers" who could do elementary arithmetic with roman numerals.

Peculiarities of Mathematical Vernacular

- **Generally:** Formulae can different grammatical roles:
  - Mathematical statements – i.e. clauses that can be true or false, e.g. $x > 5$, or $3+5=7$, or $x^2 + y^2 = z^2$.
  - Mathematical objects: 3, n, $x^2 + y^2 + z^2$, $\int_1^0 x^{3/2} dx$ (independent of their "type")

  And they need to fit into the surrounding sentence grammatically, e.g.
  - "*If $x > 0$ and $y > 0$, then $x + y > 0$.*" is OK.
  - "*If 4 then it is prime.*" is not.

Peculiarities of Mathematical Vernacular

- **Generally:** Formulae can different grammatical roles:
  - Mathematical statements – i.e. clauses that can be true or false, e.g. $x > 5$, or 3+5=7, or $x^2 + y^2 = z^2$.
  - Mathematical objects: 3, n, $x^2 + y^2 + z^2$, $\int_1^0 x^{3/2} dx$ (independent of their "type")

  And they need to fit into the surrounding sentence grammatically, e.g.
  - "*If $x > 0$ and $y > 0$, then $x + y > 0$.*" is OK.
  - "*If 4 then it is prime.*" is not.
- **Observation:** Mathematical vernacular loves to name objects/statements for precise references
  - "*There is a natural number n, such that $n^2 = 9$.*" (anaphoric)
  - "*Let $p = 3x^2 + 7x + 2342534$, then $p^3 + 17p + 1$ is irreducible.*" (saves space)
  - Definitions, theorems, example, and even equations are often numbered.
- **Example 1.14.** A mathematician would say 2. instead of 1. (normal English) (see the numbering)
  1. "*If a farmer has a donkey, he beats it with a stick*"
  2. "*If a farmer f has a donkey d, f beats d with a stick s*".

  Form 2. has the advantage that we can refer back to f, d and s from the outside. (which we cannot in the English sentence – the linguists say).

Talking about Mathematics Efficiently (Aggregation, Sequences, Ellipses)

- **Example 1.15.** Mathematical vernacular aggregates objects/statements for cognitive efficiency.
  - "$a \in S$, $b \in S$, and $c \in S \rightsquigarrow$ "$a, b, c \in S$"," (object aggregation)
  - "$i \geq 0$ and $i \leq n$" $\rightsquigarrow$ "$0 < i < n$", (statement aggregation)
  - "For all n with $n > 0 \ldots$" $\rightsquigarrow$ "For all $n > 0 \ldots$" (apposition; note seeming grammatical conflict)
- **Definition 1.16.** Mathematical vernacular uses the concept of sequences instead of lists. Sequences are usually finite (i.e. of finite length), but can be infinite as well.

Talking about Mathematics Efficiently (Aggregation, Sequences, Ellipses)

- **Example 1.20.** Mathematical vernacular aggregates objects/statements for cognitive efficiency.
  - "*$a \in S$, $b \in S$, and $c \in S$* $\leadsto$ "*$a, b, c \in S$*"," (object aggregation)
  - "*$i \geq 0$ and $i \leq n$*" $\leadsto$ "$0 < i < n$", (statement aggregation)
  - "*For all n with $n > 0$ ...*" $\leadsto$ "*For all $n > 0$ ...*" (apposition; note seeming grammatical conflict)
- **Definition 1.21.** Mathematical vernacular uses the concept of sequences instead of lists. Sequences are usually finite (i.e. of finite length), but can be infinite as well.
- **Definition 1.22.** Mathematical vernacular uses ellipses ($\ldots$) as a constructor for sequences. The meaning of an ellipsis is usually considered "obvious" and left for interpretation by the reader.
- **Example 1.23.** Ellipses allow to write down large objects easily (offload the effort to the reader)
  - $1, \ldots, n$ $\leadsto$ the sequence of natural numbers between 1 and n in order.
  - $1, 4, 9, 16, \ldots$ $\leadsto$ the sequences of squares in order
  - $e_1, \ldots, e_n$ $\leadsto$ a sequence of objects e_i for $1 < i < n$.

Talking about Mathematics Efficiently (Aggregation, Sequences, Ellipses)

- **Example 1.25.** Mathematical vernacular aggregates objects/statements for cognitive efficiency.
  - "*$a \in S$, $b \in S$, and $c \in S$* $\rightsquigarrow$ "*$a, b, c \in S$*"," (object aggregation)
  - "*$i \geq 0$ and $i \leq n$*" $\rightsquigarrow$ "$0 < i < n$", (statement aggregation)
  - "*For all n with $n > 0$ ...*" $\rightsquigarrow$ "*For all $n > 0$ ...*" (apposition; note seeming grammatical conflict)
- **Definition 1.26.** Mathematical vernacular uses the concept of sequences instead of lists. Sequences are usually finite (i.e. of finite length), but can be infinite as well.
- **Definition 1.27.** Mathematical vernacular uses ellipses ($\ldots$) as a constructor for sequences. The meaning of an ellipsis is usually considered "obvious" and left for interpretation by the reader.
- **Example 1.28.** Ellipses allow to write down large objects easily (offload the effort to the reader)
  - $1, \ldots, n \rightsquigarrow$ the sequence of natural numbers between 1 and n in order.
  - $1, 4, 9, 16, \ldots \rightsquigarrow$ the sequences of squares in order
  - $e_1, \ldots, e_n \rightsquigarrow$ a sequence of objects e_i for $1 < i < n$.
- **Argument Sequences:** Sequences are useful as argument sequences: we feed them into (flexary) constructors to create new objects.
- **Example 1.29.**
  - sets: $\{1, \ldots, n\}$, $\{1, 4, 9, 16, \ldots\}$, $S_1 \cap \ldots \cap S_n$, $S_1 \times \ldots \times S_n$, ...
  - sums, products, ...: $n_1 + \ldots + n_k$, $n_1 \cdot \ldots \cdot n_k$, ...

3.2 Talking about Mathematical Statements

Mathematical Statements & Proofs

- **Recall:** Mathematical statements are declarative sentences that can be true or false.
- Statements come in different epistemic varieties: (more on them below)
  - Definitions: statements that introduce new global identifiers for important objects
  - Assertions: statements that state properties of mathematical objects
  - Examples: statements that exhibit a witness for some property
  - Axioms: statements that characterize the objects of a certain domain of discourse or theory.

Mathematical Statements & Proofs

- **Recall:** Mathematical statements are declarative sentences that can be true or false.
- Statements come in different epistemic varieties: (more on them below)
  - Definitions: statements that introduce new global identifiers for important objects
  - Assertions: statements that state properties of mathematical objects
  - Examples: statements that exhibit a witness for some property
  - Axioms: statements that characterize the objects of a certain domain of discourse or theory.

  Definition 2.2. Mathematical assertions are pragmatically classified into categories:
-   - A lemma is an easily proved statement which is helpful for proving other propositions and theorems, but is usually not particularly interesting in its own right.
  - A proposition is a statement which is interesting in its own right,
  - A theorem is a more important statement than a proposition which says something definitive on the subject, and often takes more effort to prove than a proposition or lemma.
  - A corollary is a quick consequence of a proposition or theorem that was proven recently.
  - A conjecture is a statement that is thought to be provable, but has not been yet.

  All but the last are sometimes collectively referred to as results.

Mathematical Statements & Proofs

- **Recall:** Mathematical statements are declarative sentences that can be true or false.
- Statements come in different epistemic varieties: (more on them below)
  - Definitions: statements that introduce new global identifiers for important objects
  - Assertions: statements that state properties of mathematical objects
  - Examples: statements that exhibit a witness for some property
  - Axioms: statements that characterize the objects of a certain domain of discourse or theory.

  Definition 2.3. Mathematical assertions are pragmatically classified into categories:
- - A lemma is an easily proved statement which is helpful for proving other propositions and theorems, but is usually not particularly interesting in its own right.
  - A proposition is a statement which is interesting in its own right,
  - A theorem is a more important statement than a proposition which says something definitive on the subject, and often takes more effort to prove than a proposition or lemma.
  - A corollary is a quick consequence of a proposition or theorem that was proven recently.
  - A conjecture is a statement that is thought to be provable, but has not been yet.

  All but the last are sometimes collectively referred to as results.
- **Additionally we have:** Proofs: arguments that justify the truth of statements beyond any doubt.
- Proofs are not really statements, but we sometimes treat them together.

Talking about Mathematics (MathTalk)

- **Definition 2.4.** Abbreviations for mathematical statements in MathTalk
  - $\wedge$ and $\vee$ are common notations for "*and*" and "*or*"
  - "*not*" is in mathematical statements often denoted with $\neg$
  - $\forall x.P$ ($\forall x{\in}S.P$) stands for "*condition P holds for all x (in S)*"
  - $\exists x.P$ ($\exists x{\in}S.P$) stands for "*there exists an x (in S) such that proposition P holds*"
  - $\nexists x.P$ ($\nexists x{\in}S.P$) stands for "*there exists no x (in S) such that proposition P holds*"
  - $\exists^1 x.P$ ($\exists^1 x{\in}S.P$) stands for "*there exists one and only one x (in S) such that proposition P holds*"
  - iff as abbreviation for "*if and only if*", symbolized by "$\Leftrightarrow$"
  - the symbol "$\Rightarrow$" is used a as shortcut for "*implies*"; we can read $A \Rightarrow B$ as "*if A then B*".
- **Observation:** With these abbreviations we can use formulae for complex statements.
- **Example 2.5.** $\forall x.\exists y.x = y \Leftrightarrow \neg x \neq y$ reads

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

Peano Axioms in MathTalk

- **Example 2.6.** We can write the Peano Axioms in MathTalk: If we write $n \in \mathbb{N}_1$ for "*n is a unary natural number*", and $P(n)$ for "*n has property P*", then we can write
  - $o \in \mathbb{N}_1$ (zero is a unary natural number)
  - $\forall n \in \mathbb{N}_1 . s(n) \in \mathbb{N}_1 \wedge n \neq s(n)$ ($\mathbb{N}_1$ closed under successors, distinct)
  - $\neg(\exists n \in \mathbb{N}_1 . o = s(n))$ (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)
  - $\forall P . P(o) \wedge (\forall n \in \mathbb{N}_1 . P(n) \Rightarrow P(s(n))) \Rightarrow (\forall m \in \mathbb{N}_1 . P(m))$ (induction)

Declarations in Mathematical Vernacular

- **Example 2.7.** We often see clauses like "*Let* $\epsilon, \delta > 0$ ...", they
  - introduce new identifier ϵ and δ, (denoting new named objects that we can use later)
  - declare that ϵ and δ are "arbitrary but fixed" in the scope of the current statement, and
  - declare that they are positive – supposedly real numbers.
- **Definition 2.8.** In mathematical vernacular we call a clause that introduces new identifiers together with some properties a declaration.
- In a complex statement in mathematical vernacular, declarations can stack up to build a context of identifiers that are local to that statement.
- **Definition 2.9.** The scope of an identifier is the part of a program or expression where the reference valid; that is, where the identifier can be used to refer. In other parts of the program or expression, the identifier may refer to a different entity, or to nothing at all (it may be unbound).
- **Crucial Observation:** definitions have "global scope", declarations have "local scope".
- **Observation:** Declarations are essentially universal quantifications
  - The "*Let...*" clause in the example above is "*For all* $\epsilon, \delta > 0$, ..."
  - but declarations are grammatically clauses, so the sentence structure becomes simpler. (especially when iterated)

Dealing with conceptual complexity in Mathematical Vernacular

- **Problem:** Some concepts or objects in mathematics are inherently very complicated.
- **Coping Method:** An process of incrementally increasing complexity:
  - Start dealing with simple concepts and objects and explore their properties, understand them thoroughly by looking at examples and theorems, learn to apply them by solving problems.
  - Combine simple concepts and objects to compound ones, give them telling names, and do the same.
  - repeat the above until you reach truly interesting concepts and objects.
- **Definition 2.10.** We call the act of naming complex objects (and the sentences used for writing this down) definitions.
- Mathematics has developed various forms of definitions: definition schemata.

Definition Schemata – Simple/Pattern Definition

- **Definition 2.11.** A simple definition introduces a name (the definiendum) for a compound object or concept (the definiens).
  The definiendum must be new, i.e. may not have been used for anything else, in particular, the definiendum may not occur in the definiens. We use the symbols $:=$ (and the inverse $=:$) to write simple definitions in formulae.
- **Example 2.12.** We can give the unary natural number $////$ the name φ. In a formula we write this as $\varphi := ////$ or $//// =: \varphi$.

Definition Schemata – Simple/Pattern Definition

- **Definition 2.15.** A simple definition introduces a name (the definiendum) for a compound object or concept (the definiens).
  The definiendum must be new, i.e. may not have been used for anything else, in particular, the definiendum may not occur in the definiens. We use the symbols $:=$ (and the inverse $=:$) to write simple definitions in formulae.
- **Example 2.16.** We can give the unary natural number $////$ the name φ. In a formula we write this as $\varphi := ////$ or $//// =: \varphi$.
- A somewhat more refined form of definition is used for operators on and relations between objects.
- **Definition 2.17.** In a pattern definition the definiendum is the operator or relation is applied to n distinct variables – called pattern variables – $v_1, \ldots, v_n$ as arguments, and the definiens is an expression in these variables.
  When the new operator is applied to arguments $a_1, \ldots, a_n$, then its value is the definiens expression where the v_i are replaced by the a_i.
  We use the symbol $:=$ for operator definitions and $:\Leftrightarrow$ for relation definitions.
- **Example 2.18.** The following is a pattern definition for the set intersection operator $\cap$:

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

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

Implicit Definitions

- We now come to a very powerful definition schema.
- **Definition 2.19.** An implicit definition (also called definition by description) is an clause or expression A, such that we can prove "there is exactly one n such that A" ($\exists^1 n.\mathrm{A}$), where n is a new name – the definiendum.
  If such an unique existence proof exists, we call n well-defined.
- **Example 2.20.** $\forall x.x \in \emptyset$ is an implicit definition for the empty set $\emptyset$.
  Indeed we can prove unique existence of $\emptyset$ by just exhibiting $\{\}$ and showing that any other set S with $\forall x.x \notin S$ we have $S \equiv \emptyset$. S cannot have elements, so it has the same elements as $\emptyset$, and thus $S \equiv \emptyset$.
- **Example 2.21.** Consider the implicit definition

  The exponential function is that function $f \colon \mathbb{R} \to \mathbb{R}$ with $f' = f$ and $f(0) = 1$.

  here A is the clause "$f' = f$ and $f(0) = 1$".
  Well-definedness is mathematically non-trivial; see e.g. [here]

Mathematical Examples

- Mathematics uses examples and counterexamples to support understanding a property P:
  - examples give us a sense of the extent of P, (the set objects that satisfy C)
  - counterexample help delineate the border of P.

Mathematical Examples

- Mathematics uses examples and counterexamples to support understanding a property P:
  - examples give us a sense of the extent of P, (the set objects that satisfy C)
  - counterexample help delineate the border of P.

Definition 2.24. An example E is a mathematical statement that consists of

- - a symbol p for the exemplandum (*plural* exemplanda): the property to be exemplified,
  - the exemplans (*plural* exemplantia), an expression A denoting a mathematical object that acts as witness object for the property p, and
  - (optionally) a justification of E, i.e. a proof π that $p(a)$ holds in the current context.

  Correspondingly, in a counterexample (an example for the complement of p) π is a proof that $p(a)$ does not hold.

- **Observation:** The justification is often trivial $\rightsquigarrow$ omit, but can be very involved.

Mathematical Examples

- Mathematics uses examples and counterexamples to support understanding a property P:
  - examples give us a sense of the extent of P, (the set objects that satisfy C)
  - counterexample help delineate the border of P.

Definition 2.26. An example E is a mathematical statement that consists of

- - a symbol p for the exemplandum (*plural* exemplanda): the property to be exemplified,
  - the exemplans (*plural* exemplantia), an expression A denoting a mathematical object that acts as witness object for the property p, and
  - (optionally) a justification of E, i.e. a proof π that $p(a)$ holds in the current context.

Correspondingly, in a counterexample (an example for the complement of p) π is a proof that $p(a)$ does not hold.

- **Observation:** The justification is often trivial $\rightsquigarrow$ omit, but can be very involved.
- **Example 2.27.** The following statement is a mathematical example:

  Example 3.1.7 *(Continuous) The identity function on $\mathbb{R}$ is continuous.*

  - The exemplandum p is "continuous",
  - the exemplans A is "The identity function on $\mathbb{R}$", and
  - the justification π, a proof of continuity $\mathrm{Id}_{\mathbb{R}}$: "Let $\epsilon > 0$, then we choose $\delta := \epsilon \ldots$" is omitted.

3.3 Talking about Mathematical Proofs and Arguments

A Language for Mathematical Proofs

- Now that we understand how mathematicians talk about objects and statements, ...

A Language for Mathematical Proofs

- Now that we understand how mathematicians talk about objects and statements, ...
- ...the only things left over for mathematical vernacular is to understand how to
  - prove that a statement is indeed true
  - argue about truth while jointly developing a proof
- **Definition 3.2.** We will call the language (extension to MathTalk) that allows to do that ProofTalk.

A Language for Mathematical Proofs

- Now that we understand how mathematicians talk about objects and statements, ...
- ...the only things left over for mathematical vernacular is to understand how to
  - prove that a statement is indeed true
  - argue about truth while jointly developing a proof
- **Definition 3.3.** We will call the language (extension to MathTalk) that allows to do that ProofTalk.
- Let's look at some data to understand the phenomena involved.

ProofTalk by Example

- Say we want to prove a theorem like the following:
- **Theorem 3.4.** *There are infinitely many primes.*

- **Intuition:** ProofTalk is "arguing by the rules"! (but what are the rules?)

ProofTalk by Example

- Say we want to prove a theorem like the following:
- **Theorem 3.5.** *There are infinitely many primes.*
- **Observation:** There are many possible arguments for the truth of this statement.
  - *Proof sketch:* Euclid proved this ca. 300 BC, so we leave it as an exercise. (proof by authority)

- **Intuition:** ProofTalk is "arguing by the rules"! (but what are the rules?)

ProofTalk by Example

- Say we want to prove a theorem like the following:
- **Theorem 3.6.** *There are infinitely many primes.*
- **Observation:** There are many possible arguments for the truth of this statement.
  - *Proof sketch:* Euclid proved this ca. 300 BC, so we leave it as an exercise. (proof by authority)
  - *Proof sketch:* Suppose $p_1, p_2, \ldots, p_k$ are all the primes. Then, let $P = \prod_{i=1}^{k} p_i + 1$ and p a prime dividing P. But every p_i divides $P - 1$ so p cannot be any of them. Therefore p is a new prime. Contradiction, so there are infinitely many primes. (an informal argument)

- **Intuition:** ProofTalk is "arguing by the rules"! (but what are the rules?)

ProofTalk by Example

- Say we want to prove a theorem like the following:
- **Theorem 3.7.** *There are infinitely many primes.*
- **Observation:** There are many possible arguments for the truth of this statement.
  - *Proof sketch:* Euclid proved this ca. 300 BC, so we leave it as an exercise. (proof by authority)
  - *Proof sketch:* Suppose $p_1, p_2, \ldots, p_k$ are all the primes. Then, let $P = \prod_{i=1}^{k} p_i + 1$ and p a prime dividing P. But every p_i divides $P - 1$ so p cannot be any of them. Therefore p is a new prime. Contradiction, so there are infinitely many primes. (an informal argument)
  - *Proof:* (showing the structure of the argument)
    1. Suppose $p_1, p_2, \ldots, p_k$ are all the primes.
    2. Then, let $P := \prod_{i=1}^{k} p_i + 1$ and p a prime dividing P.
    3. But every p_i divides $P - 1$ so p cannot be any of them.
    4. Therefore p is a new prime.
    5. Contradiction, so there are infinitely many primes.

- **Intuition:** ProofTalk is "arguing by the rules"! (but what are the rules?)

ProofTalk by Example

- Say we want to prove a theorem like the following:
- **Theorem 3.8.** *There are infinitely many primes.*
- **Observation:** There are many possible arguments for the truth of this statement.
  - *Proof sketch:* Euclid proved this ca. 300 BC, so we leave it as an exercise. (proof by authority)
  - *Proof sketch:* Suppose $p_1, p_2, \ldots, p_k$ are all the primes. Then, let $P = \prod_{i=1}^{k} p_i + 1$ and p a prime dividing P. But every p_i divides $P - 1$ so p cannot be any of them. Therefore p is a new prime. Contradiction, so there are infinitely many primes. (an informal argument)
  - *Proof:* (showing the structure of the argument)
    1. Suppose $p_1, p_2, \ldots, p_k$ are all the primes.
    2. Then, let $P := \prod_{i=1}^{k} p_i + 1$ and p a prime dividing P.
    3. But every p_i divides $P - 1$ so p cannot be any of them.
    4. Therefore p is a new prime.
    5. Contradiction, so there are infinitely many primes.
  - *Proof:* We prove the assertion by contradiction (the first step above in full detail)
    1. Let us assume that the set S of primes is finite
    2. Then $\#(S) = k$ for some $k \in \mathbb{N}$.
    3. Thus $S = \{p_1, \ldots, p_k\}$ for suitable $p_i \in \mathbb{N}$.
    4. ...
- **Intuition:** ProofTalk is "arguing by the rules"! (but what are the rules?)

ProofTalk Rules – Syllogisms

- ProofTalk consists of a set of "rules of felicitous mathematical argumentation".
- **Definition 3.9.** A syllogism (also called a proof method) is an argument that applies deductive reasoning to arrive at a conclusion based on two (or more) proposition that are asserted or assumed to be true.
  Today we usually reserve the term syllogism to informal arguments; formal arguments are called inference rules.
- For formal reasoning via inference rules see GLOIN, AI-1, KRMT, ...
- Aristotle put forward a system of 13 syllogisms in his book "*Organon*" [Coo38] around 40 BC.
- ProofTalk is another system more oriented towards modern proof practice.

ProofTalk Rules – Proof by Contradiction

- **Definition 3.10.** In a proof by contradiction, we make an assumption (“not A”) to the contrary of what we want to prove, namely “A”, and then we show that the assumption leads to a contradiction and therefore must be false. That lets us to conclude that “not not A” must be true, and thus “A”.

ProofTalk Rules – Proof by Contradiction

- **Definition 3.12.** In a proof by contradiction, we make an assumption ("not A") to the contrary of what we want to prove, namely "A", and then we show that the assumption leads to a contradiction and therefore must be false. That lets us to conclude that "not not A" must be true, and thus "A".
- **Example 3.13 (Continuing from above).** In the proof above
  - the assumption "not A" is "*Suppose $p_1, p_2, \ldots, p_k$ are all the primes.*"
  - the contradiction is "*p is a new prime*", i.e not one of the p_i.

  These two cannot be true at the same time, so one must be false.
  This must be the assumption, since the contradiction was proven from it.
  So we conclude that there is no k, such that p_k is that last prime.

ProofTalk Rules – Proof by Contradiction

- **Definition 3.14.** In a proof by contradiction, we make an assumption (“not A”) to the contrary of what we want to prove, namely “A”, and then we show that the assumption leads to a contradiction and therefore must be false. That lets us to conclude that “not not A” must be true, and thus “A”.
- **Example 3.15 (Continuing from above).** In the proof above
  - the assumption “not A” is “*Suppose $p_1, p_2, \ldots, p_k$ are all the primes.*”
  - the contradiction is “*p is a new prime*”, i.e not one of the p_i.

  These two cannot be true at the same time, so one must be false.
  This must be the assumption, since the contradiction was proven from it.
  So we conclude that there is no k, such that p_k is that last prime.
- **Intuition:** We make an assumption “not A” that leads us into trouble – which is exactly where we want to be as we want to prove A.

Giving Names to Objects we know must exist

- Actually, the assumption above is that "*the set of all primes is not finite*". (to eventually show by contradition that it is infinite).
- This tells us that
  - there is (only) a finite number of prime numbers – we name it k
  - there are k prime numbers in the set – we name them p_i for $1 < i < k$.
- **Definition 3.16.** The naming rule allows to give names to objects that must exist.
- This may seem like a small thing, but it makes our (proof) life much easier, because these objects are exactly what we want to argue with.

Proving an if-then by Local Assumptions

- **Definition 3.17.** If we want to prove a statement of the form "*If A, then B*", then do that by
  - assuming A and proving B from that all we have established above.
  - after this subproof, we may not use A any more.

  We call this proof method a proof by local hypothesis.
- **Example 3.18.** We can prove "If the moon is made of green cheese, then my father will be a millionaire" by this method:

  Proof: by local hypothesis
  1. We assume that the moon is made of green cheese.
     1.1. My father has a concession to mine it.
     1.2. Green cheese is valuable, selling it makes him a million.
  3. This proves the assertion.

  □

Chaining

- **Definition 3.19.** If we know "*If A then B*" and A', and if we can turn A into A' by replacing some variables in A with concrete values, then chaining allows us to conclude B', which arises from B via the same variable replacements.
- **Example 3.20.** If we know that
  - "Socrates is a human."
  - "For all x that are human, x is also mortal."

  We can conclude "Socreates is mortal" by chaining.
- Chaining is probably the most-used ProofTalk rule of them all.

Proving a "*for all x*"

- If we want to prove "*A for all x*", then we (*A* usually contains x)
  - prove A without regard for x $\leadsto$ proof D.
  - Then argue something like "*as x was chosen arbitrarily when we proved A, we know A for every x*". (if it is indeed true that x has not been restricted).

Proof by Case Analysis

- You can sometimes prove a statement by:
  1. Dividing the situation into cases which exhaust all the possibilities; and
  2. Showing that the statement follows in all cases. (it's important to cover all the possibilities.)

Proof by Case Analysis

- You can sometimes prove a statement by:
  1. Dividing the situation into cases which exhaust all the possibilities; and
  2. Showing that the statement follows in all cases. (it's important to cover all the possibilities.)
- **Definition 3.23.** If we know that that one of the cases $A_1, \ldots, A_k$ must always hold, then the proof by cases and we can show that "if A_1 then C" holds for all $1 < i < k$, then the proof method allows us to conclude that C holds outright.
- Don't confuse this with trying examples; an example is not a proof.

Proof by Case Analysis

- You can sometimes prove a statement by:
  1. Dividing the situation into cases which exhaust all the possibilities; and
  2. Showing that the statement follows in all cases. (it's important to cover all the possibilities.)
- **Definition 3.25.** If we know that that one of the cases $A_1, \ldots, A_k$ must always hold, then the proof by cases and we can show that "if A_1 then C" holds for all $1 < i < k$, then the proof method allows us to conclude that C holds outright.
- Don't confuse this with trying examples; an example is not a proof.
- **Lemma 3.26.** *For all rational numbers a and b, if $ab = 0$, then $a = 0$ or $b = 0$.*

  Proof:
- 1. Let $a, b \in \mathbb{Q}$ and $ab = 0$.
  2. Obviously: $a = 0$ or $a \neq 0$.
  3. We prove that $a = 0$ or $b = 0$ by the two cases induced.
  4. $a = 0$
     4.1. Then the conclusion of the lemma is trivially true and so there is nothing to prove.
  6. $a \neq 0$
     6.1. We can multiply both sides of the equation $ab = 0$ by $\frac{1}{a}$ and obtain $\frac{1}{a}ab = \frac{1}{a}0$.
     6.2. Reducing the fractions gives $b = 0$.
  8. In both cases, $a = 0$ or $b = 0$, so we are done.

Without Loss of Generality

- Have you ever seen phrases like
  - "*without loss of generality we assume that p is odd*" or even
  - "*WLOG p is odd*"?
- They are weird (and very useful) idiomatic expressions that allows to simplify ProofTalk proofs.

Without Loss of Generality

- Have you ever seen phrases like
  - "*without loss of generality we assume that p is odd*" or even
  - "*WLOG p is odd*"?
- They are weird (and very useful) idiomatic expressions that allows to simplify ProofTalk proofs.
- **Example 3.28.** We want to prove $Q(p)$ for all prime numbers p. Then starting the proof with "*WLOG p is odd*" means that we can additionally assume that "*p is odd*" in the proof of $Q(p)$.

Without Loss of Generality

- Have you ever seen phrases like
  - "*without loss of generality we assume that p is odd*" or even
  - "*WLOG p is odd*"?
- They are weird (and very useful) idiomatic expressions that allows to simplify ProofTalk proofs.
- **Example 3.29.** We want to prove $Q(p)$ for all prime numbers p. Then starting the proof with "*WLOG p is odd*" means that we can additionally assume that "*p is odd*" in the proof of $Q(p)$.
  - This can be justified by
    1. In all cases where p is not odd ($\widehat{=}$ even) but still prime (so $p = 2$) prove $Q(p)$.
    2. We can prove that "*p must be even or odd*".

Without Loss of Generality

- Have you ever seen phrases like
  - "*without loss of generality we assume that p is odd*" or even
  - "*WLOG p is odd*"?
- They are weird (and very useful) idiomatic expressions that allows to simplify ProofTalk proofs.
- **Example 3.30.** We want to prove $Q(p)$ for all prime numbers p. Then starting the proof with "*WLOG p is odd*" means that we can additionally assume that "*p is odd*" in the proof of $Q(p)$.
  - This can be justified by
    1. In all cases where p is not odd ($\widehat{=}$ even) but still prime (so $p = 2$) prove $Q(p)$.
    2. We can prove that "*p must be even or odd*".
  - Indeed, if we know 2. then we can argue by cases:
    - one for p even which is just 1.
    - one for "*if p is odd then $Q(p)$*", which is left over.

Without Loss of Generality

- Have you ever seen phrases like
  - "*without loss of generality we assume that p is odd*" or even
  - "*WLOG p is odd*"?
- They are weird (and very useful) idiomatic expressions that allows to simplify ProofTalk proofs.
- **Example 3.31.** We want to prove $Q(p)$ for all prime numbers p. Then starting the proof with "*WLOG p is odd*" means that we can additionally assume that "*p is odd*" in the proof of $Q(p)$.
  - This can be justified by
    1. In all cases where p is not odd ($\widehat{=}$ even) but still prime (so $p = 2$) prove $Q(p)$.
    2. We can prove that "*p must be even or odd*".
  - Indeed, if we know 2. then we can argue by cases:
    - one for p even which is just 1.
    - one for "*if p is odd then $Q(p)$*", which is left over.
  - The main feature of WLOG is that both proofs are deemed so "easy" that we do not have to show them.

ProofTalk Non-rules – Proof by Intimidation

- **Definition 3.32.** Proof by intimidation refers to a specific form of hand-waving argument loaded with jargon and obscure results (proof by obscurity) or by marking it as obvious or trivial (proof by triviality).
  It attempts to intimidate the audience into simply accepting the result without evidence by appealing to their ignorance or lack of understanding.
- **Example 3.33.** Beware of the following indicators of proof by triviality:
  - "*Clearly...*"
  - "*It is self-evident that...*"
  - "*It can be easily shown that...*"
  - "*... does not warrant a proof.*"
  - "*The proof is left as an exercise for the reader.*"
  - "*It is trivial...*"
  - "*Trust me I am a professor, ...*"
- **Definition 3.34.** We call arguments that do not ensure the truth of the conclusion logical fallacies.
- It is important to keep ProofTalk free from logical fallacies!

ProofTalk Non-Rules – Proof by Time Travel/Circularity

- **Definition 3.35.** Circular reasoning (also known as circular logic) is a logical fallacy in which the reasoner begins with what they are trying to end with.
- In particular proof by circularity (also known as proof by time travel) is not allowed in ProofTalk.
- **Example 3.36 (Proof by Time Travel).**
  - Year 1 of course: "*Professor Dolittle will prove this theorem later in the course...*"
  - Year 2 of course: "*As you will recall, Herr Doktor Keinehilf proved this theorem in last year's classes.*"

ProofTalk Non-Rules – Proof by Time Travel/Circularity

- **Definition 3.37.** Proof by exhaustion is a method of proving that a mathematical statement is always true by working it out and showing it is true for every possible case.
- This is an extension of proof by cases to large sets of cases.
- **Definition 3.38.** Proof by examples is a logical fallacy where you check a statement A on a large (but not provably exhaustive) set of examples and use that to justify A.
- **Example 3.39 (Proof by programming).** My computer has been running for three days and has yet to find a counterexample.

ProofTalk Non-Rules – The List is Endless

- Proof by general agreement: "*All in favor?...*"
- Proof by imagination: "*Well, we'll pretend it's true...*"
- ...
- And then there are things like calculation errors. I like the following variant of reducing fractions: (even though the answer is correct, the calculation is wrong)

$$\frac{16}{64} = \frac{1\cancel{6}}{\cancel{6}4} = \frac{1}{4}$$

But this is not what really happens in practice. . .

- **Observation:** In practice we seldom see "*using proof by contradiction*" or "*by a case analysis*" . . . .
- **Even worse:** Statements are often simply claimed as "obvious" or "trivial".

But this is not what really happens in practice...

- **Observation:** In practice we seldom see "*using proof by contradiction*" or "*by a case analysis*" ....
- **Even worse:** Statements are often simply claimed as "obvious" or "trivial".
- **Claim:** There is a system behind this, which makes math communication very efficient.
- **Definition 3.41.** Proof communication (and development) is a language game between a proponent and an opponent which have the following proof moves – communicative acts that advance proofs:

Proponent		Opponent	
PC	claims A	OC	challenges claim A by counterexample C
PJ	justifies A by subproof P	ORC	requests clarification on PC or PJ
PU	cites A from the axioms, literature, or the proof so far	OA	accepts A as true or P as a well-argued subproof

where
- a PT subproof P is a sequence of PC, PJ, PU moves of any length.
- in a OC move the roles of proponent and opponent are switched; the communication restarts with claim C.

But this is not what really happens in practice...

- **Observation:** In practice we seldom see "*using proof by contradiction*" or "*by a case analysis*" ....
- **Even worse:** Statements are often simply claimed as "obvious" or "trivial".
- **Claim:** There is a system behind this, which makes math communication very efficient.
- **Definition 3.42.** Proof communication (and development) is a language game between a proponent and an opponent which have the following proof moves – communicative acts that advance proofs:

Proponent		Opponent	
PC	claims A	OC	challenges claim A by counterexample C
PJ	justifies A by subproof P	ORC	requests clarification on PC or PJ
PU	cites A from the axioms, literature, or the proof so far	OA	accepts A as true or P as a well-argued subproof

  where
  - a PT subproof P is a sequence of PC, PJ, PU moves of any length.
  - in a OC move the roles of proponent and opponent are switched; the communication restarts with claim C.
- **Research Practice:** A group of collaborators meet in front of a whiteboard the proponent puts out ideas, the others (acting as opponents) try to shoot them down. (roles switch regularly)
- Great for study groups for solving homework assignments as well.

3.4 Conclusion

Summary: Mathematical Vernacular

- If you think "*mathematical vernacular is weird!*", think again:
- **Summary:** Mathematical vernacular
  - has special language features to talk about objects, statements, and proofs.
  - has evolved to make communication about mathematics effective and efficient! (it is the best we currently have)
  - "is the language of science". (in particular for symbolic AI)
- I am not sure whether I was just riding my hobby-horse this chapter, or if this helps you better understand mathematical vernacular, ...
- **Take Home Message:** You will have to be good in understanding and producing it to succeed in symbolic AI. (most learn this by osmosis, you can study up)

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

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

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

- We will need them, when the other alphabets give out.
- BTW, we will also use the curly Roman and "Fraktur" alphabets:
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}, \mathcal{M}, \mathcal{N}, \mathcal{O}, \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$
$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}, \mathfrak{E}, \mathfrak{F}, \mathfrak{G}, \mathfrak{H}, \mathfrak{I}, \mathfrak{J}, \mathfrak{K}, \mathfrak{L}, \mathfrak{M}, \mathfrak{N}, \mathfrak{O}, \mathfrak{P}, \mathfrak{Q}, \mathfrak{R}, \mathfrak{S}, \mathfrak{T}, \mathfrak{U}, \mathfrak{V}, \mathfrak{W}, \mathfrak{X}, \mathfrak{Y}, \mathfrak{Z}$
- **Note:** Just knowing the letters is not sufficient (more work for you)
  - To understand! mathematical vernacular you need to know the letter correspondence (ν to n)
  - To talk/write mathematical vernacular, you need to pronounce and write the letters

  $\leftsquigarrow$ Having to say "*the funny Greek letter that looks a bit like a w*" is embarrassing!

Chapter 4
Elementary Discrete Math

4.1 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 ($*1845$, $\dagger 1918$))
- 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.
- **Definition 1.1 (Representations of Sets).** 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 1.2.** *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)

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

Relations between Sets

- **Definition 1.3.** set equality: $(A \equiv B) :\equiv (\forall x.x \in A \Leftrightarrow x \in B)$
- **Definition 1.4.** subset: $(A \subseteq B) :\equiv (\forall x.x \in A \Rightarrow x \in B)$
- **Definition 1.5.** proper subset: $(A \subset B) :\equiv (A \subseteq B) \wedge (A \not\equiv B)$
- **Definition 1.6.** superset: $(A \supseteq B) :\equiv (\forall x.x \in B \Rightarrow x \in A)$
- **Definition 1.7.** proper superset: $(A \supset B) :\equiv (A \supseteq B) \wedge (A \not\equiv B)$

Operations on Sets

- **Definition 1.8.** union: $A \cup B := \{x \mid x \in A \vee x \in B\}$
- **Definition 1.9.** union over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcup_{i \in I} S_i := \{x \mid \exists i \in I . x \in S_i\}$.
- **Definition 1.10.** intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$
- **Definition 1.11.** intersection over a collection: Let I be a set and S_i a family of sets indexed by I, then $\bigcap_{i \in I} S_i := \{x \mid \forall i \in I . x \in S_i\}$.
- **Definition 1.12.** set difference: $A \backslash B := \{x \mid x \in A \wedge x \notin B\}$
- **Definition 1.13.** the power set: $\mathcal{P}(A) := \{S \mid S \subseteq A\}$
- **Definition 1.14.** the empty set: $\forall x . x \notin \emptyset$
- **Definition 1.15.** Cartesian product: $A \times B := \{(a,b) \mid a \in A \wedge b \in B\}$, call (a,b) pair.
- **Definition 1.16.** n fold Cartesian product: $A_1 \times \ldots \times A_n := \{\langle a_1, \ldots, a_n \rangle \mid \forall i . 1 \leq i \leq n \Rightarrow a_i \in A_i\}$, call $\langle a_1, \ldots, a_n \rangle$ an n tuple
- **Definition 1.17.** n dim Cartesian space: $A^n := \{\langle a_1, \ldots, a_n \rangle \mid 1 \leq i \leq n \Rightarrow a_i \in A\}$, call $\langle a_1, \ldots, a_n \rangle$ a vector
- **Definition 1.18.** We write $S_1 \cup \ldots \cup S_n$ for $\bigcup_{i \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\}} S_i$ and $S_1 \cap \ldots \cap S_n$ 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 1.19.** The size $\#(A)$ of a set A is the number of elements in A.
- Intuitively we should have the following identities:
  - $\#(\{a,b,c\}) = 3$
  - $\#(\mathbb{N}) = \infty$ (infinity)
  - $\#(A \cup B) \leq \#(A) + \#(B)$ (⚠ cases with ∞)
  - $\#(A \cap B) \leq \min \#(A), \#(B)$
  - $\#(A \times B) = \#(A) \cdot \#(B)$
- 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 $\{O\}, \{\{O\}\}, \{O, \{O\}\}, \ldots \in \mathcal{S}$,
  - contains all unions, intersections, power sets,
  - contains itself: $\mathcal{S} \in \mathcal{S}$ (scary!)
- Let's make $\mathcal{S}$ less scary

A less scary $\mathcal{S}$?

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

  In either case, we have $\mathcal{S}' \in \mathcal{S}'$ iff $\mathcal{S}' \notin \mathcal{S}'$, which is a contradiction! (Russell's Antinomy [Bertrand Russell '03])
- **Does MathTalk help?:** no: $\mathcal{S}' := \{m \,|\, m \notin m\}$
  - 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)

4.2 Relations

Relations

- **Definition 2.1.** $R \subseteq A{\times}B$ is a (binary) relation between A and B.
- **Definition 2.2.** If $A = B$ then R is called a relation on A.
- **Definition 2.3.** $R \subseteq A{\times}B$ is called total iff $\forall x{\in}A.\exists y{\in}B.(x,y) \in R$.
- **Definition 2.4.** $R^{-1}{:=}\{(y,x) \mid (x,y) \in R\}$ is the converse relation of R.
- **Note:** $R^{-1} \subseteq B{\times}A$.
- **Definition 2.5.** 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 2.6.**relation $\subseteq$, $=$, *has_color*
- **Note:** We do not really need ternary, quaternary, ... relations
  - **Idea:** Consider $A \times B \times C$ as $A{\times}(B{\times}C)$ and $\langle a, b, c\rangle$ as $(a,(b,c))$
  - 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 2.7 (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 2.8.** The equality relation is an equivalence relation on any set.
- **Example 2.9.** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.

Strict and Non-Strict Partial Orders

- **Definition 2.10.** A relation $R \subseteq A \times A$ is called
  - partial ordering on A, iff R is reflexive, antisymmetric, and transitive on A.
  - strict partial ordering 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 2.11 (Linear order).** A partial ordering 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 2.12.** The $\leq$ relation is a linear order on $\mathbb{N}$ (all elements are comparable)
- **Example 2.13.** The "ancestor-of" relation is a partial order that is not linear.
- **Lemma 2.14.** *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 2.15.** *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.*

4.3 Functions

Functions (as special relations)

- **Definition 3.1.** $f \subseteq X \times Y$, 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:** $f : X \rightharpoonup Y$; $x \mapsto y$ if $(x,y) \in f$ (arrow notation)
- **Definition 3.2.** call X the domain (write $\mathrm{dom}(f)$), and Y the codomain ($\mathrm{codom}(f)$) (come with f)
- **Notation:** $f(x) = y$ instead of $(x,y) \in f$ (function application)
- **Definition 3.3.** We call a partial function $f : X \rightharpoonup Y$ undefined at $x \in X$, iff $(x,y) \notin f$ for all $y \in Y$. (write $f(x) = \bot$)
- **Definition 3.4.** If $f : X \rightharpoonup Y$ is a total relation, we call f a total function and write $f : X \to Y$. ($\forall x{\in}X.\exists^1 y \in Y.(x,y) \in f$)
  - **Notation:** $f : x \mapsto y$ if $(x,y) \in f$ (arrow notation)
- **Definition 3.5.** The identity function on a set A is defined as $\mathrm{Id}_A{:=}\{(a,a) \mid a \in A\}$.
- ⚠ **:** This probably does not conform to your intuition about functions. Do not worry, just think of them as two different things they will come together over time.(In this course we will use “function” as defined here!)

Function Spaces

- **Definition 3.6.** Given sets A and B We will call the set $A \to B$ ($A \rightharpoonup B$) of all (partial) functions from A to B the (partial) function space from A to B.
- **Example 3.7.** Let $\mathbb{B} := \{0, 1\}$ be a two-element set, then

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

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

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

Lambda-Notation for Functions

- **Problem:** In mathematics we write $f(x){:=}x^2+3x+5$ to define a function f, then we can talk about $\mathrm{dom}(f)$. But if we do not want to use a name, we can only say $\mathrm{dom}(\{(x,y)\in\mathbb{R}\times\mathbb{R}\,|\,y=x^2+3x+5\})$
- **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 $\{f_a\,|\,a\in A\}$ of functions. (is a an argument or jut a "parameter"?)
- **Definition 3.8.** To make the role of arguments extremely clear, we write functions in λ notation. For $f=\{(x,E)\,|\,x\in X\}$, where E is an expression, we write $\lambda x{\in}X.E$.

Lambda-Notation for Functions (continued)

- **Example 3.9.** The simplest function we always try everything on is the identity function:

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

- **Example 3.10.** We can also to more complex expressions, here we take the square function

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

Lambda-Notation for Functions (continued)

- **Example 3.11.** λ notation also works for more complicated domains. In this case we have pairs as arguments.

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

Properties of functions, and their converses

- **Definition 3.12.** A function $f\colon S \to T$ is called
  - injective iff $\forall x, y \in S. f(x) = f(y) \Rightarrow x = y$.
  - surjective iff $\forall y \in T. \exists x \in S. f(x) = y$.
  - bijective iff f is injective and surjective.
- **Observation 3.13.** *If f is injective, then the converse relation f^{-1} is a partial function.*
- **Observation 3.14.** *If f is surjective, then the converse f^{-1} is a total relation.*
- **Definition 3.15.** If f is bijective, call the converse relation inverse function, we (also) write it as f^{-1}.
- **Observation 3.16.** *If f is bijective, then f^{-1} is a total function.*
- **Observation 3.17.** *If $f\colon A \to B$ is bijective, then $f \circ f^{-1} = \mathrm{Id}_A$ and $f^{-1} \circ f = \mathrm{Id}_B$.*
- **Example 3.18.** The function $\nu\colon \mathbb{N}_1 \to \mathbb{N}$ with $\nu(o) = 0$ and $\nu(s(n)) = \nu(n) + 1$ is a bijection between the unary natural numbers and the natural numbers 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 3.19.** We say that a set A is finite and has cardinality $\#(A) \in \mathbb{N}$, iff there is a bijective function $f \colon A \to \{n \in \mathbb{N} \mid n < \#(A)\}$.
- **Definition 3.20.** We say that a set A is countably infinite, iff there is a bijective function $f \colon A \to \mathbb{N}$. A set is called countable, iff it is finite or countably infinite.
- **Theorem 3.21.** *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 3.22.** If $f \in A \to B$ and $g \in B \to C$ are functions, then we call

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

  the composition of g and f (read g "after" f).
- **Definition 3.23.** Let $f \in A \to B$ 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 3.24.** Let $f : A \to B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call
  - $f(A') := \{b \in B \mid \exists a \in A'.(a,b) \in f\}$ the image of A' under f,
  - $\mathrm{Im}(f) := f(A)$ the image of f, and
  - $f^{-1}(B') := \{a \in A \mid \exists b \in B'.(a,b) \in f\}$ the preimage of B' under f

4.4 Equivalence Relations and Quotients

Equivalence and Equality

- **Recap:** We have defined equivalence relation as a reflexive, symmetric, and transitive relation.
- **Example 4.1.** Equality is an equivalence relation. (trivially)
- **Example 4.2.** Let S be a set of persons, and $=_A \subseteq S^2$, such that $s =_A t$, iff s and t have the same age, then $=_A$ is an equivalence relation on S.
- **Example 4.3 (Tragic Counterexample).** Let S be a set of persons, then the relation $loves \subseteq S^2$ with $s\ loves\ t$, iff s loves t is not an equivalence relation on S.
- **Example 4.4.** Let S and T be sets and $S \sim_{\#} T$, iff there is a bijection $f : S \to T$ (we say that equinumerous), then $\sim_{\#}$ is an equivalence relation.
- **Observation 4.5.** *Equality is the most fine-grained (i.e. smallest wrt. the partial ordering $\subseteq$) equivalence relation.* (*it distinguishes most*)
- **Lemma 4.6.** *If S is a set and $R \subseteq S^2$ is an equivalence relation, then $= \subseteq R$.*
- **Idea:** Sometimes we want to lump together objects if they are in a given equivalence relation.

Let's make this Concept Mathematical

- **Definition 4.7.** Let S be a set and R be an equivalence relation on S, then for any $x \in S$ we call the set $[x]_R {:=} \{y \in S \,|\, R(x,y)\}$ the equivalence class of x (under R), and the set $S/R {:=} \{[x]_R \,|\, x \in S\}$ the quotient space of S (under R), it is often read as S "modulo R". The element x is called the representative of $[x]_R \in S/R$.
- **Definition 4.8.** The mapping $\pi_R : S \to S/R\, ; x \mapsto [x]_R$ is called the canonical projection or canonical surjection of S to S/R.
- **Definition 4.9.** Let R be an equivalence relation on S, a subset $M \subseteq S$ is called a system of representatives, iff M contains exactly one representative for each equivalence class of of R.
- **Observation:** Often a quotient spaces S/R behaves similarly to the original set S.
- **Example 4.10.** Remember: $n \equiv_k m$, iff $k|(n-m)$. It is an equivalence relation and $\pi_{\equiv_k} : \mathbb{Z}/\equiv_k \to \{n \,|\, 0 \leq n \leq (k-1)\}$ is bijective.
- **Lemma 4.11.** *For the equality relation $=$ on a set S, then $[x]_= \in S/=$ is always a singleton and thus $S \sim_{\#} S/=$.*

Chapter 5
Computing with Functions over Inductively Defined Sets

5.1 Standard ML: A Functional Programming Language

Enough theory, let us start computing with functions

- We will use Standard ML (SML) in this course.
- **Definition 1.1.** 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:** There are multiple tutorials.
- **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 1.2 (Predefined objects in SML).** (SML comes with a basic inventory)
  - basic types `int`, `real`, `bool`, `string` , . . .
  - basic type constructors $->$, $*$,
  - basic operators numbers, `true`, `false`, $+$, $*$, $-$, $>$, ^, . . . (overloading)
  - control structures **if** Φ **then** E_1 **else** E_2;
  - comments (*this is a comment *)

Programming in SML (Declarations)

- **Definition 1.3.** 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 1.4.** A local declaration uses **let** to bind variables in its scope (delineated by **in** and **end**).
- **Example 1.5.** 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 1.6.** Using structured patterns, we can declare more than one variable. We call this pattern matching.
- **Example 1.7 (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 1.8.** Anonymous variables (if we are not interested in one value)

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

- **Example 1.9.** 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 $\widehat{=}$ empty list, :: $\widehat{=}$ list constructor "cons")

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

- A simple recursive function: creating integer intervals

```
- fun upto (m,n) = if m>n then nil else m::upto(m+1,n);
val upto = fn : int * int -> int list
- upto(2,5);
val it = [2,3,4,5] : int list
```

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

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.
- **Definition 1.10.** We write $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots \rightsquigarrow t_n$ for tracing the recursive arguments t_i through a recursive computation.
- **Example 1.11.** We have the following evaluation trace with result [2,3,4]

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

- **Definition 1.12.** We call an SML function recursive, iff the function is called in the function definition.
- **Example 1.13.** Note that recursive functions need not terminate, consider the function

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

which has the evaluation sequence

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

Defining Functions by cases

- **Idea:** Use the fact that lists are either nil or of the form X::Xs, where X is an element and Xs is a list of elements.
- The body of an SML function can be made of several cases separated by the operator |.
- **Example 1.14.** 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"]];
val it = ["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)
  - 'Erika feuert nur untreue Fakire' (for German speakers)

Higher-Order Functions

- **Idea:** Pass functions as arguments (functions are normal values.)
- **Example 1.15.** Mapping a function over a list

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

- **Example 1.16.** We can program the map function ourselves!

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

- **Example 1.17.** Declaring functions (yes, functions are normal values.)

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

- **Example 1.18.** Returning functions: (again, functions are normal values.)

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

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 1.20.** $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ with $+((3,2))$ instead of $+(3,2)$

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

- **Workaround 2:** Make use of functions as results.
- **Example 1.21.** $+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ withn $+(3)(2)$ instead of $+((3,2))$.

```
- fun cascaded_plus (x:int) = (fn y:int => x + y);
val it = 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 1.22.** Call a function Cartesian, iff the argument type is a product type, call it cascaded, iff the result type is a function type.
- **Example 1.23.** The following function is both Cartesian and cascading

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

- **Convenient:** Bracket elision conventions
  - $e_1\ e_2\ e_3 \leadsto (e_1\ e_2)\ e_3$ (function application associates to the left)
  - $\tau_1 \texttt{->} \tau_2 \texttt{->} \tau_3 \leadsto \tau_1 \texttt{->} (\tau_2 \texttt{->} \tau_3)$ (function types associate to the right)
- SML uses these elision rules

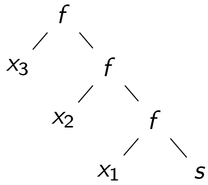
```
- fun both_plus (x:int,y:int) = fn (z:int) => x + y + z;
Val both_plus int * int -> int -> int
```

- Another simplification (related to those above)

```
- cascaded_plus 4 5;
val it = 9 : int
```

Folding Operators

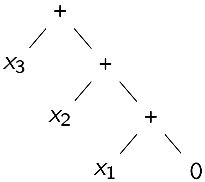
- **Definition 1.24.** SML provides the left folding operator to realize a recurrent computation schema



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

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

- **Example 1.25.** Folding the iterator **op+** with start value 0:



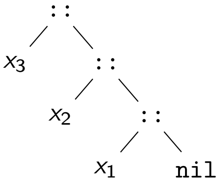
$$\text{foldl } \mathbf{op}{+}\ 0\ [x_1,x_2,x_3] = x_3{+}(x_2{+}(x_1{+}0))$$

Thus the function given by the expression foldl **op**+ 0 adds the elements of integer lists.

Folding Procedures (continued)

- **Example 1.26 (Reversing Lists).**

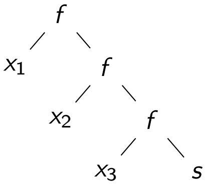
$$\text{foldl } \mathbf{op}{::} \text{ nil } [x_1,x_2,x_3] = x_3 :: (x_2 :: (x_1 :: \text{nil}))$$



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

Folding Procedures (foldr)

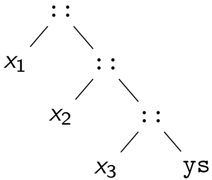
- **Definition 1.27.** The right folding operator foldr is a variant of foldl that processes the list elements in reverse order.




```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

foldr f s $[x_1,x_2,x_3]$ = f(x_1,f(x_2,f(x_3,s)))

- **Example 1.28 (Appending Lists).**



foldr **op**:: ys $[x_1,x_2,x_3] = x_1 :: (x_2 :: (x_3 :: \mathrm{ys}))$

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)

5.2 Inductively Defined Sets and Computation

What about Addition, is that a function?

- **Problem:** Addition takes two arguments (binary function)
- **One solution:** $+ \colon \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ is unary
- **Definition 2.1 (Defining equations).** $+((n,o)) = n$ (base) and $+((m,s(n))) = s(+((m,n)))$ (step)
- **Theorem 2.2.** $+ \subseteq \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$ *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 2.3.** *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)

  we have two cases

  1. base case $(m = o)$
     1.1. choose $l := n$, so we have $+((n,o)) = n = l$.
     1.2. For any $l' = +((n,o))$, we have $l' = n = l$.
  3. induction step $(m = s(k))$
     3.1. assume that there is a unique $r = +((n,k))$, choose $l := s(r)$, so we have $+((n,s(k))) = s(+((n,k))) = s(r)$.
     3.2. Again, for any $l' = +((n,s(k)))$ we have $l' = l$.

  $\square$

- **Corollary 2.4.** $+ \colon \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ *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 $s \colon \mathbb{N}_1 \to \mathbb{N}_1$
- **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 2.5.** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$
- **Example 2.6.** 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 2.7.** An inductively defined set $\langle S, C\rangle$ is a set S together with a finite set $C := \{c_i \mid 1 \leq i \leq n\}$ of k_i ary constructors $c_i \colon S^{k_i} \to S$ with $k_i \geq 0$, such that
  - if $s_i \in S$ for all $1 \leq i \leq k_i$, then $c_i(s_1, \ldots, k_i) \in S$ (generated by constructors)
  - all constructors are injective, (no internal confusion)
  - $\mathrm{Im}(c_i) \cap \mathrm{Im}(c_j) = \emptyset$ for $i \neq j$, and (no confusion between constructors)
  - for every $s \in S$ there is a constructor $c \in C$ with $s \in \mathrm{Im}(c)$. (no junk)
- Note that we also allow nullary constructors here.
- **Example 2.8.** $\langle \mathbb{N}_1, \{s, o\}\rangle$ is an inductively defined set.
- *Proof:* We check the three conditions in 2.7 using the Peano Axioms
  1. Generation is guaranteed by P1 and P2
  2. Internal confusion is prevented P4
  3. Inter-constructor confusion is averted by P3
  4. Junk is prohibited by P5.

□

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

- **Lists of (unary) natural numbers:** $[1,2,3]$, $[7,7]$, $[]$, ...
  - nil-rule: start with the empty list $[]$
  - cons-rule: extend the list by adding a number $n \in \mathbb{N}_1$ at the front
- **Definition 2.9.** two constructors: $\mathrm{nil} \in \mathcal{L}[\mathbb{N}]$ and $\mathrm{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$
- **Example 2.10.** e.g. $[3,2,1] \mathrel{\widehat{=}} \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, \mathrm{nil})))$ and $[] \mathrel{\widehat{=}} \mathrm{nil}$
- **Definition 2.11.** We will call the following set of axioms are called the list Peano axioms for $\mathcal{L}[\mathbb{N}]$ in analogy to the Peano Axioms in 1.12.
- **Axiom 2.12 (LP1).** $\mathrm{nil} \in \mathcal{L}[\mathbb{N}]$ *(generation axiom (nil))*
- **Axiom 2.13 (LP2).** $\mathrm{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ *(generation axiom (cons))*
- **Axiom 2.14 (LP3).** nil *is not a* cons*-value*
- **Axiom 2.15 (LP4).** cons *is injective*
- **Axiom 2.16 (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 $\mathrm{cons}(n,l)$ has property P*

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

Operations on Lists: Append

- **Definition 2.17.** The append function $@ \colon \mathcal{L}[\mathbb{N}] \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ concatenates lists
  Defining equations: $\mathrm{nil}@l = l$ and $\mathrm{cons}(n, l)@r = \mathrm{cons}(n, l@r)$
- **Example 2.18.** $[3, 2, 1]@[1, 2] = [3, 2, 1, 1, 2]$ and $[]@[1, 2, 3] = [1, 2, 3] = [1, 2, 3]@[]$
- **Lemma 2.19.** *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?)
  we have two cases
  1. base case: $l = \mathrm{nil}$
     1.1. must have $s = r$.
  3. induction step: $l = \mathrm{cons}(n, k)$ for some list k
     3.1. Assume that here is a unique s' with $s' = k@r$,
     3.2. then $s = \mathrm{cons}(n, k)@r = \mathrm{cons}(n, k@r) = \mathrm{cons}(n, s')$.

  □
- **Corollary 2.20.** *Append is a function* (*see, this just worked fine!*)

Operations on Lists: more examples

- **Definition 2.21.** $\lambda(\mathrm{nil}) = o$ and $\lambda(\mathrm{cons}(n, l)) = s(\lambda(l))$
- **Definition 2.22.** $\rho(\mathrm{nil}) = \mathrm{nil}$ and $\rho(\mathrm{cons}(n, l)) = \rho(l)@\mathrm{cons}(n, \mathrm{nil})$.

5.3 Inductively Defined Sets in SML

Data Type Declarations I

- **Definition 3.1.** SML data types 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 | zero
```

  this gives us constructor functions zero : mynat and suc : mynat −> 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 -> 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)

- **Example 3.7.** A type for weekdays (nullary constructors)

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

- **Example 3.8.** Use it as basis for rule-based procedure (first clause takes precedence)

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

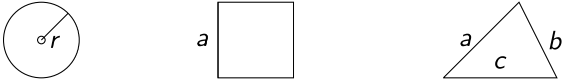
This give us

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

Data Types Example (Areas of Shapes)

- **Example 3.9.** 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
```

- **New Construct:** Standard structure Math (see [Sml])
- Some experiments

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

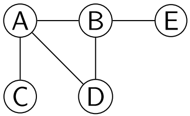
Chapter 6
Graphs and Trees

Basic Definitions: Graphs

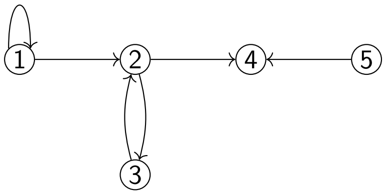
- **Definition 0.1.** An undirected graph is a pair $\langle V, E\rangle$ such that
  - V is a set of vertices (or nodes), (draw as circles)
  - $E \subseteq \{\{v, v'\} \mid v, v' \in V \wedge (v \neq v')\}$ is the set of its undirected edges. (draw as lines)
- **Definition 0.2.** A directed graph (also called digraph) is a pair $\langle V, E\rangle$ such that
  - V is a set of vertices
  - $E \subseteq V \times V$ is the set of its directed edges
- **Definition 0.3.** Given a graph $\langle V, E\rangle$. The indegree $\mathrm{indeg}(v)$ and the outdegree $\mathrm{outdeg}(v)$ (or branching factor) of a vertex $v \in V$ are defined as
  - $\mathrm{indeg}(v) = \#(\{w \mid (w,v) \in E\})$
  - $\mathrm{outdeg}(v) = \#(\{w \mid (v,w) \in E\})$
- **Note:** For an undirected graph, $\mathrm{indeg}(v) = \mathrm{outdeg}(v)$ for all nodes v.

Examples

- **Example 0.4.** An undirected graph $G_1 = \langle V_1, E_1 \rangle$, where $V_1 = \{A, B, C, D, E\}$ and $E_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{B, E\}\}$



- **Example 0.5.** A directed graph $G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{(1,1), (1,2), (2,3), (3,2), (2,4), (5,4)\}$



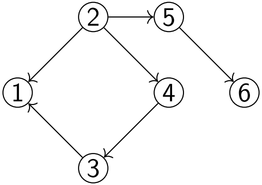
The Graph Diagrams are not Graphs

They are pictures of graphs (of course!)

Directed Graphs

- **Idea:** Directed graphs are nothing else than relations.
- **Definition 0.6.** Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$
  - initial, iff there is no $w \in V$ such that $(w,v) \in E$. (no predecessor)
  - terminal, iff there is no $w \in V$ such that $(v,w) \in E$. (no successor)

  In a graph G, node v is also called a source (sink) of G, iff it is initial (terminal) in G.
- **Example 0.7.** The node 2 is initial, and the nodess 1 and 6 are terminal in

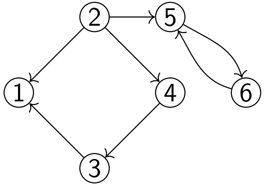


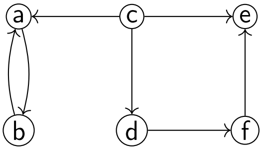
Graph Isomorphisms

- **Definition 0.8.** A isomorphism between two graphs $G = \langle V, E\rangle$ and $G' = \langle V', E'\rangle$ is a bijective function $\psi\colon V \to V'$ with

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

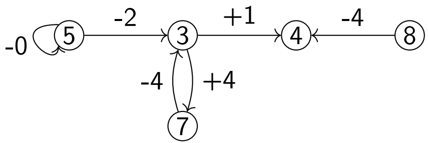
- **Definition 0.9.** Two graphs G and G' are equivalent iff there is an isomorphism ψ between G and G'.
- **Example 0.10.** G_1 and G_2 are equivalent as there exists a isomorphism $\psi := \{a \mapsto 5, b \mapsto 6, c \mapsto 2, d \mapsto 4, e \mapsto 1, f \mapsto 3\}$ between them.





Labeled Graphs

- **Definition 0.11.** A labeled graph G is a quadruple $\langle V, E, L, l\rangle$ where $\langle V, E\rangle$ is a graph and $l\colon V \cup E \rightarrow L$ is a partial function into a set L of labels.
- **Notation:** Write labels next to their vertex or edge. If the actual name of a vertex does not matter, its label can be written into it.
- **Example 0.12.** $G = \langle V, E, L, l\rangle$ with $V = \{A, B, C, D, E\}$, where
  - $E = \{(A,A), (A,B), (B,C), (C,B), (B,D), (E,D)\}$
  - $l\colon V \cup E \rightarrow \{+, -, \emptyset\} \times \{1, \ldots, 9\}$ with
    - $l(A) = 5$, $l(B) = 3$, $l(C) = 7$, $l(D) = 4$, $l(E) = 8$,
    - $l((A,A)) = -0$, $l((A,B)) = -2$, $l((B,C)) = +4$,
    - $l((C,B)) = -4$, $l((B,D)) = +1$, $l((E,D)) = -4$

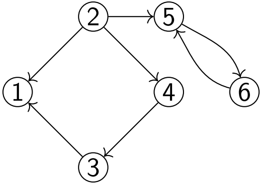


Paths in Graphs

- **Definition 0.13.** Given a graph $G := \langle V, E\rangle$ we call a $n+1$-tuple $p=\langle v_0, \ldots, v_n\rangle \in V^{n+1}$ a path in G iff $(v_{(i-1)}, v_i) \in E$ for all $1 \leq i \leq n$ and $n > 0$.
  - We say that the v_i are nodes on p and that v_0 and v_n are linked by p.
  - v_0 and v_n are called the start and end of p (write $\mathrm{start}(p)$ and $\mathrm{end}(p)$), the other v_i are called inner nodes of p.
  - n is called the length of p (write $\mathrm{len}(p)$).
  - We denote the set of paths in G with $\Pi(G)$
- **Note:** Not all v_i-s in a path are necessarily different.
- **Notation:** For a graph $G = \langle V, E\rangle$ and a path $p = \langle v_1, \ldots, v_n\rangle \in V^{n+1}$, write
  - $v \in p$, iff $v \in V$ is a vertex on the path $\quad (\exists i. v_i = v)$
  - $e \in p$, iff $e = (v, v') \in E$ is an edge on the path $\quad (\exists i. v_i = v \wedge v_{i+1} = v')$
- **Notation:** We write $\Pi(G)$ for the set of all paths in a graph G.

Cycles in Graphs

- **Definition 0.14.** Given a directed graph $\langle V, E\rangle$, a path p is called cyclic (or a cycle) iff $\mathrm{start}(p) = \mathrm{end}(p)$. A cycle $\langle v_0, \ldots, v_n\rangle$ is called simple, iff $v_i \neq v_j$ for $1 \leq i, j \leq n$ with $i \neq j$.
- **Example 0.15.** $(2,4), (4,3)$ and $(2,5), (5,6), (6,5), (5,6), (6,5)$ are paths in

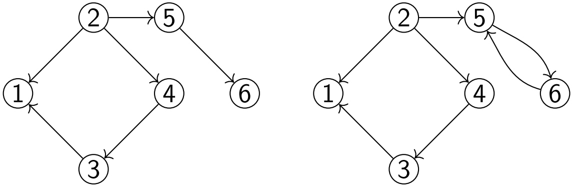


$(2,4), (4,3), (3,1), (1,2)$ is not a path (no edge from vertex 1 to vertex 2)
The graph is not acyclic ($(5,6), (6,5)$ is a cycle)

- **Definition 0.16.** We will sometimes use the abbreviation DAG for "directed acyclic graph".

Graph Depth

- **Definition 0.17.** Let $\langle V, E\rangle$ be a directed graph, then the depth $\mathrm{dp}(v)$ of a vertex $v \in V$ is defined to be 0, if v is a source of G and $\sup(\{\mathrm{len}(p) \mid \mathrm{indeg}(\mathrm{start}(p)) = 0 \wedge \mathrm{end}(p) = v\})$ otherwise, i.e. the length of the longest path from a source of G to v. (can be infinite).
- **Definition 0.18.** Given a digraph $G = \langle V, E\rangle$. The depth ($\mathrm{dp}(G)$) of G is defined as $\sup(\{\mathrm{len}(p) \mid p \in \Pi(G)\})$, i.e. the maximal path length in G.
- **Example 0.19.** The vertex 6 has depth two in the left graph and infinite depth in the right one.

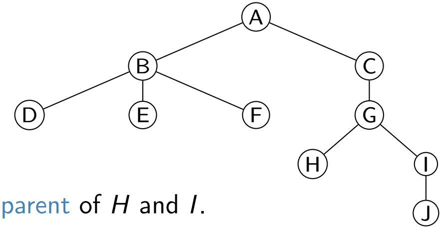


The left graph has depth three (cf. node 1), the right one has infinite depth (cf. nodes 5 and 6)

Trees

- **Definition 0.20.** A tree is a DAG $G = \langle V, E\rangle$ such that
  - There is exactly one initial node $v_r \in V$ (called the root)
  - All nodes but the root have indegree 1.

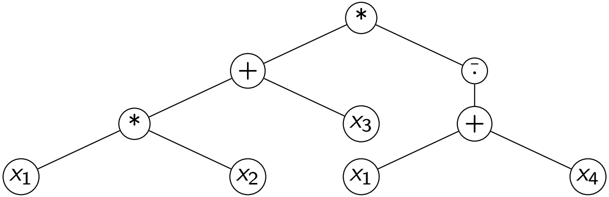
  We call v the parent of w, iff $(v,w) \in E$ (w is a child of v). We call a node v a leaf of G, iff it is terminal, i.e. if it does not have children.
- **Example 0.21.** A tree with root A and leaves D, E, F, H, and J.



  F is a child of B and G is the parent of H and I.
- **Lemma 0.22.** *For any node $v \in V$ except the root v_r, there is exactly one path $p \in \Pi(G)$ with $\mathrm{start}(p) = v_r$ and $\mathrm{end}(p) = v$.* (*proof by induction on the number of nodes*)

The Parse-Tree of a Boolean Expression

- **Definition 0.23.** The parse tree P_e of a Boolean expression e is a labeled tree $P_e = \langle V_e, E_e, f_e \rangle$, which is recursively defined as
  - if $e = \overline{e'}$ then $V_e := V_{e'} \cup \{v\}$, $E_e := E_{e'} \cup \{(v, v_r')\}$, and $f_e := f_{e'} \cup \{v \mapsto \overline{\cdot}\}$, where $P_{e'} = (V_{e'}, E_{e'}, f_{e'})$ is the parse tree of e', v_r' is the root of $P_{e'}$, and v is an object not in $V_{e'}$.
  - if $e = e_1 \circ e_2$ with $\circ \in \{*, +\}$ then $V_e := V_{e_1} \cup V_{e_2} \cup \{v\}$, $E_e := E_{e_1} \cup E_{e_2} \cup \{(v, v_r^1), (v, v_r^2)\}$, and $f_e := f_{e_1} \cup f_{e_2} \cup \{v \mapsto \circ\}$, where the $P_{e_i} = (V_{e_i}, E_{e_i}, f_{e_i})$ are the parse trees of e_i and v_r^i is the root of P_{e_i} and v is an object not in $V_{e_1} \cup V_{e_2}$.
  - if $e \in (V \cup C_{\mathrm{bool}})$ then, $V_e = \{e\}$ and $E_e = \emptyset$.
- **Example 0.24.**
  The parse tree of $(x_1 * x_2 + x_3) * \overline{x_1 + x_4}$ is



Chapter 7
Recap: Formal Languages and Grammars

The Mathematics of Strings

- **Definition 0.1.** An alphabet A is a finite set; we call each element $a \in A$ a character, and an n tuple $s \in A^n$ a string (of length n over A).
- **Definition 0.2.** 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:** We will often write a string $\langle c_1, \ldots, c_n\rangle$ as $"\mathtt{c_1 \ldots c_n}"$, for instance $"\mathtt{abc}"$ for $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}\rangle$
- **Example 0.3.** Take $A = \{\mathtt{h}, \mathtt{1}, /\}$ as an alphabet. Each of the members $\mathtt{h}$, $\mathtt{1}$, and $/$ is a character. The vector $\langle /, /, \mathtt{1}, \mathtt{h}, \mathtt{1}\rangle$ is a string of length 5 over A.
- **Definition 0.4 (String Length).** Given a string s we denote its length with $|s|$.
- **Definition 0.5.** The concatenation $\mathrm{conc}(s,t)$ of two strings $s = \langle s_1, \ldots, s_n\rangle \in A^n$ and $t = \langle t_1, \ldots, t_m\rangle \in A^m$ is defined as $\langle s_1, \ldots, s_n, t_1, \ldots, t_m\rangle \in A^{n+m}$.
  We will often write $\mathrm{conc}(s,t)$ as $s + t$ or simply st
- **Example 0.6.** $\mathrm{conc}("\mathtt{text}", "\mathtt{book}") = "\mathtt{text}" + "\mathtt{book}" = "\mathtt{textbook}"$

Formal Languages

- **Definition 0.7.** Let A be an alphabet, then we define the sets $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ of nonempty string and $A^* := A^+ \cup \{\epsilon\}$ of strings.
- **Example 0.8.** If $A = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, then $A^* = \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{aa}, \mathrm{ab}, \mathrm{ac}, \mathrm{ba}, \ldots, \mathrm{aaa}, \ldots\}$.
- **Definition 0.9.** A set $L \subseteq A^*$ is called a formal language over A.
- **Definition 0.10.** We use $\mathrm{c}^{[n]}$ for the string that consists of the character c repeated n times.
- **Example 0.11.** $\#^{[5]} = \langle \#, \#, \#, \#, \# \rangle$
- **Example 0.12.** The set $M := \{\mathrm{ba}^{[n]} \mid n \in \mathbb{N}\}$ of strings that start with character b followed by an arbitrary numbers of a's is a formal language over $A = \{\mathrm{a}, \mathrm{b}\}$.
- **Definition 0.13.** Let $L_1, L_2, L \subseteq \Sigma^*$ be formal languages over Σ.
  - Intersection and union: $L_1 \cap L_2$, $L_1 \cup L_2$.
  - Language complement L: $\overline{L} := \Sigma^* \backslash L$.
  - The language concatenation of L_1 and L_2: $L_1 L_2 := \{uw \mid u \in L_1,\ w \in L_2\}$. We often use $L_1 L_2$ instead of $L_1 L_2$.
  - Language power L: $L^0 := \{\epsilon\}$, $L^{n+1} := L L^n$, where $L^n := \{\mathtt{w_1} \ldots \mathtt{w_n} \mid w_i \in L,\ \text{for } i = 1 \ldots n\}$, (for $n \in \mathbb{N}$).
  - language Kleene closure L: $L^* := \bigcup_{n \in \mathbb{N}} L^n$ and also $L^+ := \bigcup_{n \in \mathbb{N}^+} L^n$.
  - The reflection of a language L: $L^R := \{w^R \mid w \in L\}$.

Phrase Structure Grammars (Theory)

- **Recap:** A formal language is an arbitrary set of symbol sequences.
- **Problem:** This may be infinite and even undecidable even if A is finite.
- **Idea:** Find a way of representing formal languages with structure finitely.
- **Definition 0.14.** A phrase structure grammar (also called type 0 grammar, unrestricted grammar, or just grammar) is a tuple $\langle N, \Sigma, P, S\rangle$ where
  - N is a finite set of nonterminal symbols,
  - Σ is a finite set of terminal symbols, members of $\Sigma \cup N$ are called symbols.
  - P is a finite set of production rules: pairs $p := h \rightarrow b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the head of p and b the body.
  - $S \in N$ is a distinguished symbol called the start symbol (also sentence symbol).

  The sets N and Σ are assumed to be disjoint. Any word $w \in \Sigma^*$ is called a terminal word.
- **Intuition:** Production rules map strings with at least one nonterminal to arbitrary other strings.
- **Notation:** If we have n rules $h \rightarrow b_i$ sharing a head, we often write $h \rightarrow b_1 \mid \ldots \mid b_n$ instead.

Phrase Structure Grammars (cont.)

- **Example 0.15.** A simple phrase structure grammar G:

$$\begin{array}{rcl} S & \rightarrow & NP\ Vi \\ NP & \rightarrow & Article\ N \\ Article & \rightarrow & \mathrm{the} \mid \mathrm{a} \mid \mathrm{an} \\ N & \rightarrow & \mathrm{dog} \mid \mathrm{teacher} \mid \ldots \\ Vi & \rightarrow & \mathrm{sleeps} \mid \mathrm{smells} \mid \ldots \end{array}$$

  Here S, is the start symbol, NP, $Article$, N, and Vi are nonterminals.

- **Definition 0.16.** A production rule whose head is a single non-terminal and whose body consists of a single terminal is called lexical or a lexical insertion rule.
  Definition 0.17. The subset of lexical rules of a grammar G is called the lexicon of G and the set of body symbols the vocabulary (or alphabet). The nonterminals in their heads are called lexical categories of G.

- **Definition 0.18.** The non-lexicon production rules are called structural, and the nonterminals in the heads are called phrasal or syntactic categories.

Phrase Structure Grammars (Theory)

- **Idea:** Each symbol sequence in a formal language can be analyzed/generated by the grammar.
- **Definition 0.19.** Given a phrase structure grammar $G:=\langle N,\Sigma,P,S\rangle$, we say G derives $t\in(\Sigma\cup N)^*$ from $s\in(\Sigma\cup N)^*$ in one step, iff there is a production rule $p\in P$ with $p=h\rightarrow b$ and there are $u,v\in(\Sigma\cup N)^*$, such that $s=suhv$ and $t=ubv$. We write $s\rightarrow^p_G t$ (or $s\rightarrow_G t$ if p is clear from the context) and use $\rightarrow^*_G$ for the reflexive transitive closure of $\rightarrow_G$. We call $s\rightarrow^*{}_G t$ a G derivation of t from s.
- **Definition 0.20.** Given a phrase structure grammar $G:=\langle N,\Sigma,P,S\rangle$, we say that $s\in(N\cup\Sigma)^*$ is a sentential form of G, iff $S\rightarrow^*{}_G s$. A sentential form that does not contain nontermials is called a sentence of G, we also say that G accepts s. We say that G rejects s, iff it is not a sentence of G.
- **Definition 0.21.** The language $\mathrm{L}(G)$ of G is the set of its sentences. We say that $\mathrm{L}(G)$ is generated by G.

  Definition 0.22. We call two grammars equivalent, iff they have the same languages.

  Definition 0.23. A grammar G is said to be universal if $\mathrm{L}(G)=\Sigma^*$.
- **Definition 0.24.** Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in a formal or a natural language by means of a grammar.

Phrase Structure Grammars (Example)

- **Example 0.25.** In the grammar G from 0.15:
  1. *Article* teacher *Vi* is a sentential form,

$$\begin{array}{lll} S & \rightarrow_G & \mathit{NP}\ \mathit{Vi} \\ & \rightarrow_G & \mathit{Article}\ N\ \mathit{Vi} \\ & \rightarrow_G & \mathit{Article}\ \text{teacher}\ \mathit{Vi} \end{array}$$

  2. "*The teacher sleeps*" is a sentence.

$$\begin{array}{lll} S & \rightarrow_G^* & \mathit{Article}\ \text{teacher}\ \mathit{Vi} \\ & \rightarrow_G & \text{the teacher}\ \mathit{Vi} \\ & \rightarrow_G & \text{the teacher sleeps} \end{array}$$

$$\begin{array}{rcl} S & \rightarrow & \mathit{NP}\ \mathit{Vi} \\ \mathit{NP} & \rightarrow & \mathit{Article}\ N \\ \mathit{Article} & \rightarrow & \text{the} \mid \text{a} \mid \text{an} \mid \ldots \\ N & \rightarrow & \text{dog} \mid \text{teacher} \mid \ldots \\ \mathit{Vi} & \rightarrow & \text{sleeps} \mid \text{smells} \mid \ldots \end{array}$$

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the “size” of its language.
- **Definition 0.26.** We call a grammar:
  1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
  2. context-free (or type 2), if the heads have exactly one symbol,
  3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

  By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the "size" of its language.
- **Definition 0.30.** We call a grammar:
  1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
  2. context-free (or type 2), if the heads have exactly one symbol,
  3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

  By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.
- **Example 0.31 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$ is accepted by

$$
\begin{aligned}
S &\to \mathrm{a\,b\,c} \mid A \\
A &\to \mathrm{a}\,A\,B\,\mathrm{c} \mid \mathrm{a\,b\,c} \\
\mathrm{c}\,B &\to B\,\mathrm{c} \\
\mathrm{b}\,B &\to \mathrm{b\,b}
\end{aligned}
$$

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the "size" of its language.
- **Definition 0.34.** We call a grammar:
  1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
  2. context-free (or type 2), if the heads have exactly one symbol,
  3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

  By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.
- **Example 0.35 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$
- **Example 0.36 (Context-free).** The language $\{a^{[n]}b^{[n]}\}$ is accepted by $S \rightarrow \mathrm{a}\ S\ \mathrm{b} \mid \epsilon$.

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the "size" of its language.
- **Definition 0.38.** We call a grammar:
  1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
  2. context-free (or type 2), if the heads have exactly one symbol,
  3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

  By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.
- **Example 0.39 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$
- **Example 0.40 (Context-free).** The language $\{a^{[n]}b^{[n]}\}$
- **Example 0.41 (Regular).** The language $\{a^{[n]}\}$ is accepted by $S \rightarrow S$ a

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the "size" of its language.
- **Definition 0.42.** We call a grammar:
  1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
  2. context-free (or type 2), if the heads have exactly one symbol,
  3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

  By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.
- **Example 0.43 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$
- **Example 0.44 (Context-free).** The language $\{a^{[n]}b^{[n]}\}$
- **Example 0.45 (Regular).** The language $\{a^{[n]}\}$
- **Observation:** Natural languages are probably context-sensitive but parsable in real time! (like languages low in the hierarchy)

Useful Extensions of Phrase Structure Grammars

- **Definition 0.46.** The Bachus Naur form or Backus normal form (BNF) is a metasyntax notation for context-free grammars.
  It extends the body of a production rule by mutiple (admissible) constructors:
  - alternative: $s_1 \mid \ldots \mid s_n$,
  - repetition: s^* (arbitrary many s) and s^+ (at least one s),
  - optional: $[s]$ (zero or one times),
  - grouping: $(s_1 ; \ldots ; s_n)$, useful e.g. for repetition,
  - character sets: $[s{-}t]$ (all characters c with $s \leq c \leq t$ for a given ordering on the characters), and
  - complements: $[^\wedge s_1, \ldots, s_n]$, provided that the base alphabet is finite.
- **Observation:** All of these can be eliminated, .e.g ($\leadsto$ many more rules)
  - replace $X \to Z\ (s^*)\ W$ with the production rules $X \to Z\ Y\ W$, $Y \to \epsilon$, and $Y \to Y\ s$.
  - replace $X \to Z\ (s^+)\ W$ with the production rules $X \to Z\ Y\ W$, $Y \to s$, and $Y \to Y\ s$.

An Grammar Notation for SMAI

- **Problem:** In grammars, notations for nonterminal symbols should be
  - short and mnemonic (for the use in the body)
  - close to the official name of the syntactic category (for the use in the head)
- In SMAI we will only use context-free grammars (simpler, but problem still applies)
- **in SMAI:** I will try to give "grammar overviews" that combine those, e.g. the grammar of first-order logic.

variables	X	$\in$	$\mathcal{V}_1$	
function constants	f^k	$\in$	Σ_k^f	
predicate constants	p^k	$\in$	$\Sigma^p{}_k$	
terms	t	$::=$	X	variable
		$\mid$	f^0	constant
		$\mid$	$f^k(t_1, \ldots, t_k)$	application
formulae	A	$::=$	$p^k(t_1, \ldots, t_k)$	atomic
		$\mid$	$\neg\mathrm{A}$	negation
		$\mid$	$\mathrm{A}_1 \wedge \mathrm{A}_2$	conjunction
		$\mid$	$\forall X.\mathrm{A}$	quantifier

Chapter 8
Term Languages and Abstract Grammars

Term Languages

- In most applications of symbolic AI, the formal languages are very structured.
- **Example 0.1 (Arithmetic Expressions).** Consider the grammar G and the G-derivation

$$\begin{array}{rcl} \underline{\mathrm{term}} & ::= & \underline{\mathrm{num}} \mid \underline{\mathrm{var}} \mid \underline{\mathrm{sum}} \mid \underline{\mathrm{prod}} \mid \underline{\mathrm{neg}} \\ \underline{\mathrm{num}} & ::= & \underline{\mathrm{digit}^*} \\ \underline{\mathrm{digit}} & ::= & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0 \\ \underline{\mathrm{var}} & ::= & '\mathtt{X}' ; \underline{\mathrm{num}} \\ \underline{\mathrm{sum}} & ::= & '(' ; \underline{\mathrm{term}} ; '+' ; \underline{\mathrm{term}} ')' \\ \underline{\mathrm{prod}} & ::= & '(' ; \underline{\mathrm{term}} ; '*' ; \underline{\mathrm{term}} ')' \\ \underline{\mathrm{neg}} & ::= & '-(' ; \underline{\mathrm{term}} ; ')' \end{array}$$

$$\begin{array}{rll} \underline{\mathrm{term}} & \to_G & '-(' ; \underline{\mathrm{term}} ; ')' \\ & \to_G & '-((' ; \underline{\mathrm{term}} ; '*' ; \underline{\mathrm{term}} ; '))' \\ & \to_G & '-(((' ; \underline{\mathrm{term}} ; '+' ; \underline{\mathrm{term}} ')*' ; \underline{\mathrm{num}} ; '))' \\ & \to_G & '-(((' ; \underline{\mathrm{var}} ; '+' ; \underline{\mathrm{num}} ')*' ; \underline{\mathrm{digit}} ; \underline{\mathrm{digit}} ; \underline{\mathrm{digit}} ; '))' \\ & \to_G & '-(((\mathtt{X}' ; \underline{\mathrm{num}} ; '+' ; \underline{\mathrm{digit}} ')*555))' \\ & \to_G & '-(((\mathtt{X}' ; \underline{\mathrm{digit}} ; \underline{\mathrm{digit}} ; '+3)*555))' \\ & \to_G & '-(((\mathtt{X}29+3)*555))' \end{array}$$

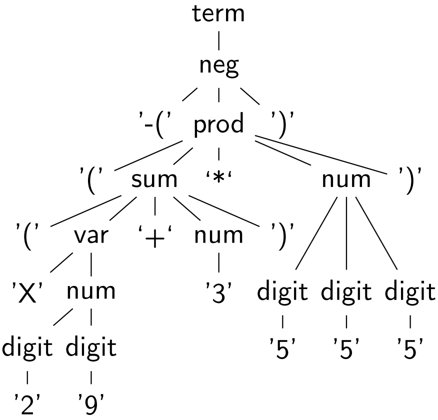
  G accepts the string $-(((X29+3)*555))$ but not $-(X29*3($.
- **Definition 0.2.** We will call such languages term languages.
- **Observation:** Strings in $L(G)$ are "well-bracketed" and can be split into sub-strings from $L(G)$.
- **Onion Principle:** The derivation peels off one layer of structure at a time down to the terminals.
- **Intuition:** The parse trees are the primary obects for symbolic AI, the strings are just the technical I/O.
- We will make a lot of use of this idea. (next)

Parse Trees of Formulae

- **Definition 0.3.** Let G be a CFG that accept a string s. Then a parse tree is an edge labeled, ordered tree P_s that represents the syntactic structure of s.
- **Problem:** There may be multiple derivations that accept a string s in a CFG.
- **Solution:** Define the parse tree for the derivation that accepts s instead.
- **Definition 0.4.** Let G be a context-free grammar and $D := s{\rightarrow^*}_G t$ a G-derivation where t is a sentence of G. We define the parse tree P_D of D recursively:
  - If $D = s \rightarrow^p_G t$, then p is a lexical rule $s \rightarrow t$ and t consists of a single terminal. Then P_D is the tree whose root has label s with one child labeled with t.
  - If $D = s \rightarrow^p_G s' \rightarrow^*_G t$, $D' := s'{\rightarrow^*}_G t$, and $p = h \rightarrow a_1 \ldots a_n$, then the root of P_D has label h and it has children are the parse trees for the subderivations for the a_i in D'.
- Term languages are ones that have enough structural characters so that the parse tree is "obvious".

- **Example 0.5.** A parse tree for (the derivation on the left for) the string $-(((X29+3)*555))$

$$\begin{array}{lll}
\underline{\text{term}} & \to_G & {}'-({}' \,;\, \underline{\text{term}} \,;\, {}'){}' \\
 & \to_G & {}'-(({}' \,;\, \underline{\text{term}} \,;\, {}'*{}' \,;\, \underline{\text{term}} \,;\, {}'))' \\
 & \to_G & {}'-((({}' \,;\, \underline{\text{term}} \,;\, {}'+{}' \,;\, \underline{\text{term}}{}')*{}' \,;\, \underline{\text{num}} \,;\, {}'))' \\
 & \to_G & {}'-((({}' \,;\, \underline{\text{var}} \,;\, {}'+{}' \,;\, \underline{\text{num}}{}')*{}' \,;\, \underline{\text{digit}} \,;\, \underline{\text{digit}} \,;\, \underline{\text{digit}} \,;\, {}'))' \\
 & \to_G & {}'-(((\mathtt{X}{}' \,;\, \underline{\text{num}} \,;\, {}'+{}' \,;\, \underline{\text{digit}}{}')*555))' \\
 & \to_G & {}'-(((\mathtt{X}{}' \,;\, \underline{\text{digit}} \,;\, \underline{\text{digit}} \,;\, {}'+3)*555))' \\
 & \to_G & {}'-(((\mathtt{X}29+3)*555))'
\end{array}$$



- This is a lot of shuffling of characters around, and we are only interested in the tree structure anyways.
- In particular, the brackets are only needed for specifing the parse tree structu re.

Programming with Arithmetic Expressions in SML

- **Example 0.6.** A data type for arithmetic expressions

```
datatype aterm = anum of int (* numbers *)
               | avar of int (* variables *)
               | aneg of aterm (* negative *)
               | asum of aterm * aterm (* sums *)
               | aprod of aterm * aterm (* products *)
```

  We can express arithmetic expressions as SML expression directly, e.g. −(((X29−3)∗555)):

```
val ex2 = aneg(aprod (asum(avar 29, anum 3), anum 555))
```

- **Note that** the aterm data type is recursive (it uses itself in contructor arguments)
- **Example 0.7 (Term Traversal).** We can program recursively with SML arithmetic expressions:

```
fun aprint (anum n) = Int.toString n
  | aprint (avar n) = "X" ^ Int.toString n
  | aprint (aneg t) = "−(" ^ aprint t ^ ")"
  | aprint (asum(t1,t2)) = "(" ^ aprint t1 ^ "+" ^ aprint t2 ^ ")"
  | aprint (aprod(t1,t2)) = "(" ^ aprint t1 ^ "*" ^ aprint t1 ^ ")"
```

  Testing on our example: aprint ex2 $\rightsquigarrow$ "−(((X29+3)∗(X29+3)))"

Computation via Tree Traversal

- We can also do more complex tasks via tree traversal:
- **Example 0.8 (Evaluation of Arithmetic Expressions).** To evaluate $-(X_{29} + 3) * 555$, we need to know the value of the variable X_{29}. This is traditionally given by an assignment that assigns values to variables, e.g. $X_1 \mapsto 3, \ldots, X_{29} \mapsto 5$, which we can represent as a list of pairs in SML:

```
type assignment = (int * aterm) list
```

Then we can represent the evaluation function by recursion over the structure of the of the expression:

```
fun aeval (anum n,a:assignment) = n
  | aeval (aneg(n),a) = ~(aeval(n,a))
  | aeval (asum(s,t),a) = aeval(s,a) + aeval(t,a)
  | aeval (aprod(s,t),a) = aeval(s,a) * aeval(t,a)
```

In all cases, the assignment is just passed on to the recursivel call. The only exception is the variable case, where we need to (recursively) find the right value from the assignment:

```
  | aeval (avar(n),(m,r)::l) = if n = m then aeval(r,l) else aeval(avar(n),l)
```

Regular Tree Grammars

- **Observation:** The situation for arithmetic expressions in SML from 0.6 is typical for term languages.
  - Inductive data types, one constructor per production rule
  - Expressions as tree-structured data structures (no brackets needed)
  - All computations on expressions via recursive tree traversal. (see ???)
- **Definition 0.9.** A tree grammar is a grammar where the result data type is trees, not strings.
- **Intuition:** If we interpret the brackets as "parse tree constructors", then we do not have to worry about matching them. ($\leadsto$ simpler grammar; usually regular)
- The string representation – with e.g. infix notations for operators – is just a derived/secondary artifact for storage/communication. ($\widehat{=}$ implementation detail)
- **Definition 0.10.** The underlying tree grammar of a language is often called the abstract grammar, and any derived (string) grammars – there may be multiple – a concrete grammar.
- We are mostly interested in abstract grammars in symbolic AI.

Standardizing Expression Trees for Symbolic AI

- **Observation:** The arithmetic expressions in 0.6 consist of
  - literals like 555 (a leaf in the parse tree)
  - named variables like X29 (another)
  - operator applications like X29+3 (+ labels an inner node).
- Other languages also have
  - named constants like π (a leaf in the parse tree)
  - binding expressions like $\forall x.P$ in MathTalk, where the bound variable x can be consistently renamed. $(\forall x.q(x) = \forall y.q(y))$
- **Idea:** We only need a tree grammar with non-terminals for these four!

  var(x) $\rightarrow$ x

  const(c) $\rightarrow$ c

  app(f,$t_1,\ldots,t_k$) $\longrightarrow$ f

  $t_1 \cdots t_n$

  bind($\beta[v_1,\ldots,v_l]t_1,\ldots,t_k$) $\rightarrow$ β

  b $\quad t_1 \cdots t_n$

  $v_1 \cdots v_n$

-

Chapter 9
Mathematical Language Recap

Mathematical Structures

- **Observation:** Mathematicians often cast classes of complex objects as mathematical structures.
- We have just seen an example of a mathematical structure: (repeated here for convenience)
- **Definition 0.1.** A phrase structure grammar (also called type 0 grammar, unrestricted grammar, or just grammar) is a tuple $\langle N, \Sigma, P, S\rangle$ where
  - N is a finite set of nonterminal symbols,
  - Σ is a finite set of terminal symbols, members of $\Sigma \cup N$ are called symbols.
  - P is a finite set of production rules: pairs $p := h \rightarrow b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the head of p and b the body.
  - $S \in N$ is a distinguished symbol called the start symbol (also sentence symbol).

  The sets N and Σ are assumed to be disjoint. Any word $w \in \Sigma^*$ is called a terminal word.
- **Intuition:** All grammars share structure: they have four components, which again share struccture, which is further described in the definition above.
- **Observation:** Even though we call production rules "pairs" above, they are also mathematical structures $\langle h, b\rangle$ with a funny notation $h \rightarrow b$.

Mathematical Structures in Programming

- **Observation:** Most programming languages have some way of creating "named structures". Referencing components is usually done via "dot notation".

Mathematical Structures in Programming

- **Observation:** Most programming languages have some way of creating "named structures". Referencing components is usually done via "dot notation".
- **Example 0.4 (Structs in C).** C data structures for representing grammars:

```
struct grule {
  char[][] head;
  char[][] body;
}
struct grammar {
  char[][] nterminals;
  char[][] termininals;
  grule[] grules;
  char[] start;
}
int main() {
  struct grule r1;
  r1.head = "foo";
  r1.body = "bar";
}
```

Mathematical Structures in Programming

- **Observation:** Most programming languages have some way of creating "named structures". Referencing components is usually done via "dot notation".
- **Example 0.6 (Structs in C).** C data structures for representing grammars:

```
struct grule {
  char[][] head;
  char[][] body;
}
struct grammar {
  char[][] nterminals;
  char[][] termininals;
  grule[] grules;
  char[] start;
}
int main() {
  struct grule r1;
  r1.head = "foo";
  r1.body = "bar";
}
```

- **Example 0.7 (Classes in OOP).** Classes in object-oriented programming languages are based on the same ideas as mathematical structures, only that OOP adds powerful inheritance mechanisms.

In SMAI we use a mixture between Math and Programming Styles

- In SMAI we use mathematical notation, ...
- **Definition 0.8.** A structure signature combines the components, their "types", and accessor names of a mathematical structure in a tabular overview.
- **Example 0.9.**

$$\text{grammar} = \left\langle \begin{array}{lll} N & \text{Set} & \text{nonterminal symbols}, \\ \Sigma & \text{Set} & \text{terminal symbols}, \\ P & \{h \rightarrow b \mid \ldots\} & \text{production rules}, \\ S & N & \text{start symbol} \end{array} \right\rangle$$

$$\text{production rule} \quad h \rightarrow b = \left\langle \begin{array}{lll} h & (\Sigma \cup N)^*, N, (\Sigma \cup N)^* & \text{head}, \\ b & (\Sigma \cup N)^* & \text{body} \end{array} \right\rangle$$

  Read the first line "N Set nonterminal symbols" in the structure above as "N is in an (unspecified) set and is a nonterminal symbol".
  Here – and in the future – we will use Set for the class of sets $\leadsto$ "N is a set".
- I will try to give structure signatures where necessary.

Chapter 10
Recap: Complexity Analysis in AI?

Performance and Scaling

- Suppose we have three algorithms to choose from. (which one to select)
- Systematic analysis reveals performance characteristics.
- **Example 0.1.** For a computational problem of size n 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?

- $2^{10} = 1\,024$ ($1024\mu s \simeq 1ms$)
- $2^{45} = 35\,184\,372\,088\,832$ ($3.5{\times}10^{13}\mu s \simeq 3.5{\times}10^{7}s \simeq 1.1Y$)
- **Example 0.2.** We denote all times that are longer than the age of the universe with —

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

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 0.6.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.
  Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 0.9.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.
  Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.
- Time/space complexity depends on size measures. (no canonical one)

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 0.12.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.
  Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.
- Time/space complexity depends on size measures. (no canonical one)
- **Definition 0.13.** The following sets are often used for S in $T(\alpha)$:

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

  where $\mathcal{O}(g) = \{f \mid \exists k > 0. f \leq_a k \cdot g\}$ and $f \leq_a g$ (f is asymptotically bounded by g), iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 0.15.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.
  Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.
- Time/space complexity depends on size measures. (no canonical one)
- **Definition 0.16.** The following sets are often used for S in $T(\alpha)$:

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

  where $\mathcal{O}(g) = \{f \mid \exists k > 0. f \leq_a k \cdot g\}$ and $f \leq_a g$ (f is asymptotically bounded by g), iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.
- **Lemma 0.17 (Growth Ranking).** *For $k' > 2$ and $k > 1$ we have*

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

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 0.18.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.
  Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.
- Time/space complexity depends on size measures. (no canonical one)
- **Definition 0.19.** The following sets are often used for S in $T(\alpha)$:

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

  where $\mathcal{O}(g) = \{f \mid \exists k > 0. f \leq_a k \cdot g\}$ and $f \leq_a g$ (f is asymptotically bounded by g), iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.
- **Lemma 0.20 (Growth Ranking).** *For $k' > 2$ and $k > 1$ we have*

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

- **For SMAI:** I expect that given an algorithm, you can determine its complexity class. (next)

Advantage: Big-Oh Arithmetics

- **Practical Advantage:** Computing with Landau sets is quite simple. (good simplification)
- **Theorem 0.21 (Computing with Landau Sets).**
  1. *If* $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$ *for any constant* $c \in \mathbb{N}$. (*drop constant factors*)
  2. *If* $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, *then* $\mathcal{O}(f + g) = \mathcal{O}(g)$. (*drop low-complexity summands*)
  3. *If* $\mathcal{O}(f \cdot g) = \mathcal{O}(f) \cdot \mathcal{O}(g)$. (*distribute over products*)
- These are not all of "big-Oh calculation rules", but they're enough for most purposes
- **Applications:** Convince yourselves using the result above that
  - $\mathcal{O}(4n^3 + 3n + 7^{1000n}) = \mathcal{O}(2^n)$
  - $\mathcal{O}(n) \subset \mathcal{O}(n \cdot \log_2(n)) \subset \mathcal{O}(n^2)$

Determining the Time/Space Complexity of Algorithms

- **Definition 0.22.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: can be accessed in constant time

Determining the Time/Space Complexity of Algorithms

- **Definition 0.23.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.24.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: need the complexity of the value

Determining the Time/Space Complexity of Algorithms

- **Definition 0.25.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.26.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: compose the complexities of the function and the argument

Determining the Time/Space Complexity of Algorithms

- **Definition 0.27.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.28.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: has to compute the value $\rightsquigarrow$ has its complexity

Determining the Time/Space Complexity of Algorithms

- **Definition 0.29.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.30.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: has the maximal complexity of the components

Determining the Time/Space Complexity of Algorithms

- **Definition 0.31.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: If α is $\varphi \,;\, \psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.32.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v,S)$.
  - composition: If α is $\varphi\,;\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
  - branching: has the maximal complexity of the condition and branches

Determining the Time/Space Complexity of Algorithms

- **Definition 0.33.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha){\in}\mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha){\in}\mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi){\in}\mathcal{O}(f)$ and $T_{\Gamma\cup C_\Gamma(\varphi)}(\psi){\in}\mathcal{O}(g)$, then $T_\Gamma(\alpha){\in}\mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma\cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v{:=}\varphi$ with $T_\Gamma(\varphi){\in}S$, then $T_\Gamma(\alpha){\in}S$ and $C_\Gamma(\alpha) = \Gamma \cup (v,S)$.
  - composition: If α is $\varphi\,;\psi$, with $T_\Gamma(\varphi){\in}P$ and $T_{\Gamma\cup C_\Gamma(\psi)}(\psi){\in}Q$, then $T_\Gamma(\alpha){\in}\max\,\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma\cup C_\Gamma(\psi)}(\psi)$.
  - branching: If α is **if**γ**then**φ**else**ψ**end**, with $T_\Gamma(\gamma){\in}C$, $T_{\Gamma\cup C_\Gamma(\gamma)}(\varphi){\in}P$, $T_{\Gamma\cup C_\Gamma(\gamma)}(\varphi){\in}Q$, and then $T_\Gamma(\alpha){\in}\max\,\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma\cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma\cup C_\Gamma(\gamma)}(\psi)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.34.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: If α is $\varphi\,;\,\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
  - branching: If α is **if**γ**then**φ**else**ψ**end**, with $T_\Gamma(\gamma) \in C$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in P$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in Q$, and then $T_\Gamma(\alpha) \in \max\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\psi)$.
  - looping: multiplies complexities

Determining the Time/Space Complexity of Algorithms

- **Definition 0.35.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: If α is $\varphi\,;\,\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
  - branching: If α is $\mathbf{if}\gamma\mathbf{then}\varphi\mathbf{else}\psi\mathbf{end}$, with $T_\Gamma(\gamma) \in C$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in P$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in Q$, and then $T_\Gamma(\alpha) \in \max\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\psi)$.
  - looping: If α is $\mathbf{while}\gamma\mathbf{do}\varphi\mathbf{end}$, with $T_\Gamma(\gamma) \in \mathcal{O}(f)$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f(n) \cdot g(n))$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi)$.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.36.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: If α is $\varphi\,;\,\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
  - branching: If α is $\mathbf{if}\gamma\mathbf{then}\varphi\mathbf{else}\psi\mathbf{end}$, with $T_\Gamma(\gamma) \in C$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in P$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in Q$, and then $T_\Gamma(\alpha) \in \max\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\psi)$.
  - looping: If α is $\mathbf{while}\gamma\mathbf{do}\varphi\mathbf{end}$, with $T_\Gamma(\gamma) \in \mathcal{O}(f)$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f(n) \cdot g(n))$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi)$.
  - The time complexity $T(\alpha)$ is just $T_\emptyset(\alpha)$, where $\emptyset$ is the empty function.

Determining the Time/Space Complexity of Algorithms

- **Definition 0.37.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the
  - time complexity $T_\Gamma(\alpha)$ of program α and
  - the context $C_\Gamma(\alpha)$ introduced by α

  by joint induction on the structure of α:
  - constant: If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
  - variable: If $\alpha = v$ with $v \in \mathrm{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
  - application: If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
  - assignment: If α is $v := \varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v, S)$.
  - composition: If α is $\varphi\,;\,\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
  - branching: If α is $\mathbf{if}\gamma\mathbf{then}\varphi\mathbf{else}\psi\mathbf{end}$, with $T_\Gamma(\gamma) \in C$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in P$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in Q$, and then $T_\Gamma(\alpha) \in \max\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\psi)$.
  - looping: If α is $\mathbf{while}\gamma\mathbf{do}\varphi\mathbf{end}$, with $T_\Gamma(\gamma) \in \mathcal{O}(f)$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f(n) \cdot g(n))$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi)$.
  - The time complexity $T(\alpha)$ is just $T_\emptyset(\alpha)$, where $\emptyset$ is the empty function.
- Recursion is much more difficult to analyze $\leadsto$ recurrences and Master's theorem.

Why Complexity Analysis? (General)

- **Example 0.38.** Once upon a time I was trying to invent an efficient algorithm.
  - My first algorithm attempt didn't work, so I had to try harder.

Why Complexity Analysis? (General)

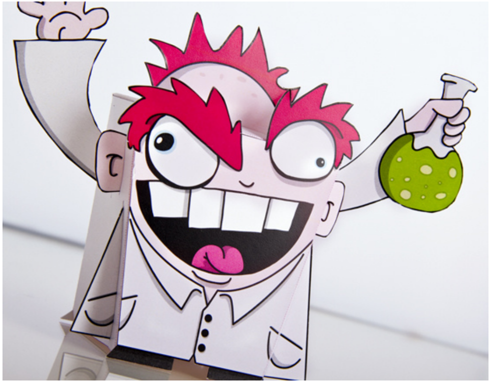
- **Example 0.39.** Once upon a time I was trying to invent an efficient algorithm.
  - My first algorithm attempt didn't work, so I had to try harder.
  - But my 2nd attempt didn't work either, which got me a bit agitated.

Why Complexity Analysis? (General)

- **Example 0.40.** Once upon a time I was trying to invent an efficient algorithm.
  - My first algorithm attempt didn't work, so I had to try harder.
  - But my 2nd attempt didn't work either, which got me a bit agitated.
  - The 3rd attempt didn't work either...

Why Complexity Analysis? (General)

- **Example 0.41.** Once upon a time I was trying to invent an efficient algorithm.
  - My first algorithm attempt didn't work, so I had to try harder.
  - But my 2nd attempt didn't work either, which got me a bit agitated.
  - The 3rd attempt didn't work either. . .
  - And neither the 4th. But then:

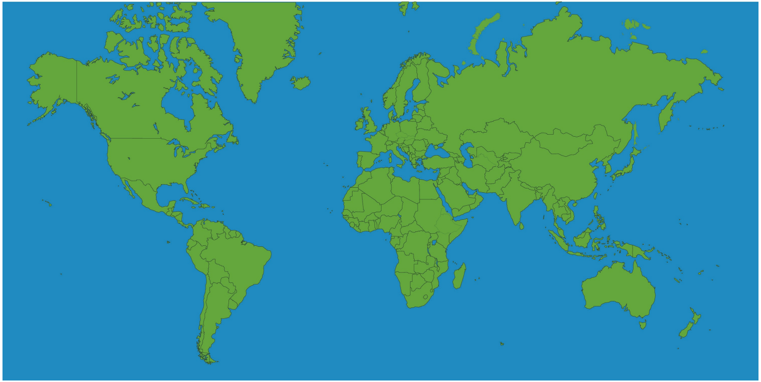


Why Complexity Analysis? (General)

- **Example 0.42.** Once upon a time I was trying to invent an efficient algorithm.
  - My first algorithm attempt didn't work, so I had to try harder.
  - But my 2nd attempt didn't work either, which got me a bit agitated.
  - The 3rd attempt didn't work either...
  - And neither the 4th. But then:
  - Ta-da ... when, for once, I turned around and looked in the other direction– CAN one actually solve this efficiently? – NP-hard was there to rescue me.

Why Complexity Analysis? (General)

- **Example 0.43.** Trying to find a sea route east to India (from Spain) (does not exist)



- **Observation:** Complexity theory saves you from spending lots of time trying to invent algorithms that do not exist.

Reminder (?): NP and PSPACE (details $\rightsquigarrow$ e.g. [GJ79])

- **Turing Machine:** Works on a tape consisting of cells, across which its Read/Write head moves. The machine has internal states. There is a Turing machine program that specifies – given the current cell content and internal state – what the subsequent internal state will be, how what the R/W head does (write a symbol and/or move). Some internal states are accepting.
- Decision problems are in NP if there is a non deterministic Turing machine that halts with an answer after time polynomial in the size of its input. Accepts if *at least one* of the possible runs accepts.
- Decision problems are in NPSPACE, if there is a non deterministic Turing machine that runs in space polynomial in the size of its input.
- **NP vs. PSPACE:** Non-deterministic polynomial space can be simulated in deterministic polynomial space. Thus PSPACE $=$ NPSPACE, and hence (trivially) NP $\subseteq$ PSPACE.
  It is commonly believed that NP$\not\supseteq$PSPACE. (similar to P $\subseteq$ NP)

The Utility of Complexity Knowledge (NP-Hardness)

- **Assume:** In 3 years from now, you have finished your studies and are working in your first industry job. Your boss Mr. X gives you a problem and says "*Solve It!*". By which he means, "*write a program that solves it efficiently*".
- **Question:** Assume further that, after trying in vain for 4 weeks, you got the next meeting with Mr. X. How could knowing about NP-hard problems help?

The Utility of Complexity Knowledge (NP-Hardness)

- **Assume:** In 3 years from now, you have finished your studies and are working in your first industry job. Your boss Mr. X gives you a problem and says "*Solve It!*". By which he means, "*write a program that solves it efficiently*".
- **Question:** Assume further that, after trying in vain for 4 weeks, you got the next meeting with Mr. X. How could knowing about NP-hard problems help?
- **Answer:** It helps you save your skin with (theoretical computer) science!
  - Do you want to say "*Um, sorry, but I couldn't find an efficient solution, please don't fire me*"?
  - Or would you rather say "*Look, I didn't find an efficient solution. But neither could all the Turing-award winners out there put together, because the problem is NP-hard*"?

References I

[Cho65] Noam Chomsky. *Syntactic structures*. Den Haag: Mouton, 1965.

[Coo38] Harold Percy Cooke, ed. *Aristotle: The Organon, Volume 1*. Vol. 391. Loeb classical library. Harvard University Press, 1938.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

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

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

[Nor+18a] Emily Nordmann et al. *Lecture capture: Practical recommendations for students and lecturers*. 2018. URL: https://osf.io/huydx/download.

[Nor+18b] Emily Nordmann et al. *Vorlesungsaufzeichnungen nutzen: Eine Anleitung für Studierende*. 2018. URL: https://osf.io/e6r7a/download.

[Pal] *Neil/Fred's Gigantic List of Palindromes*. http://www.derf.net/palindromes/. URL: http://www.derf.net/palindromes/.

References II

[Pau91] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

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

[Sml] *The Standard ML Basis Library*. 2010. URL: `http://www.standardml.org/Basis/`.