Symbolic Methods for AI

Prof. Dr. Michael Kohlhase

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

2025-05-06

0.1 Preface

0.1.1 This Document

This document contains the lecture notes for the course "*Symbolic Methods for Artificial Intelligence*" (SMAI) held at FAU Erlangen-Nürnberg in the Summer Semesters 2025 ff.

This course introduces students to the scientific methods used in Symbolic AI. It is geared towards closing the gap many students experience between their engineering-oriented undergraduate education and the abstract mathematical methods needed in the courses in the Symbolic AI Pillar of the Master AI at FAU (e.g. the AI-1 course).

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

Caveat: This document is primarily made available for the students of the SMAI course only. After multiple iterations of this course it is reasonably feature-complete, but will evolve and be polished in coming academic years.

Knowledge Representation Experiment: This document is also an experiment in knowledge representation. Under the hood, it uses the sTEX package [Koh08; sTeX], a TEX/LATEX extension for semantic markup, which allows to export the contents into active documents that adapt to the reader and can be instrumented with services based on the explicitly represented meaning of the documents.

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

0.1.2 Acknowledgments

Most of the KWARC group has contributed to the SMAI course materials in various forms; in particular:

Contents

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

In this chapter, we want to get all the organizational matters out of the way, so that we can get course contents unencumbered. We will talk about the necessary administrative details, go into how students can get most out of the course, talk about where the various resources provided with the course can be found, and finally introduce the ALeA system, an experimental – using AI methods – learning support system for the SMAI course.

1.1 Administrative Ground Rules

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

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)

Now we come to a topic that is always interesting to the students: the grading scheme.

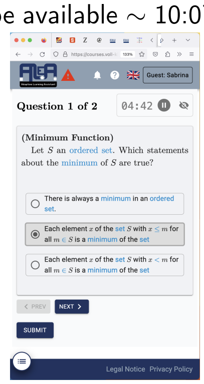
Assessment, Grades

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

- **Exam:** exam conducted in presence on paper! (~ October 8. 2025)
- **Retake Exam:** 60 minutes exam six months later. (~ 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. (~ 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 prepquizzes 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 ~ 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 1.1.1.** 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)

And now a serious warning: If you think that you can – just because SMAI is a "preparatory course" – get the 2.5 ECTS that you still need for your symbolic AI pillar without putting in the work, then you should think again.

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

In this section we will discuss a couple of measures that students may want to consider to get most out of the SMAI course.

None of the things discussed in this section – homeworks, tutorials, study groups, and attendance – are mandatory (we cannot force you to do them; we offer them to you as learning opportunities), but most of them are very clearly correlated with success (i.e. passing the exam and getting a good grade), so taking advantage of them may be in your own interest.

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 ⇝ 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)

It is very well-established experience that without doing the homework assignments (or something similar) on your own, you will not master the concepts, you will not even be able to ask sensible questions, and take very little home from the course. Just sitting in the course and nodding is not enough!

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!)

Michael Kohlhase: SMAI 8 2025-05-06

If you have questions please make sure you discuss them with the instructor, the teaching assistants, or your fellow students. There are three sensible venues for such discussions: online in the lectures, in the tutorials, which we discuss now, or in the course forum – see below. Finally, it is always a very good idea to form study groups with your friends.

Collaboration

▷ **Definition 1.2.1.** 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)

Michael Kohlhase: SMAI 9 2025-05-06

As we said above, almost all of the components of the SMAI course are optional. That even applies to attendance. But make no mistake, attendance is important to most of you. Let me explain, . . .

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!

FAU has issued a very insightful guide on using lecture videos. It is a good idea to heed these recommendations, even if they seem annoying at first.

Practical recommendations on Lecture Videos

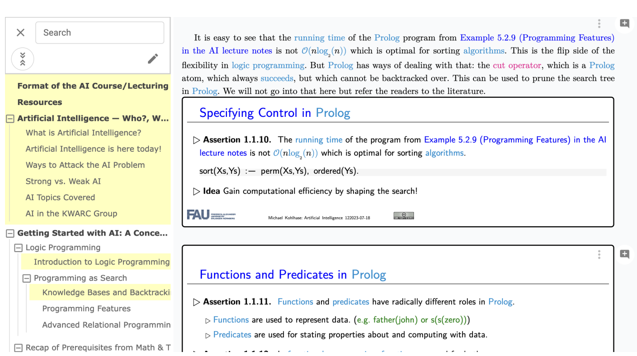
▷ **Excellent Guide:** [Nor+18a] (German version at [Nor+18b])

1.3.1 ALeA – AI-Supported Learning

In this subsection we introduce the ALEA (Adaptive Learning Assistant) system, a learning support system we will use to support students in SMAI.

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 1.3.1.** 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 1.3.2 (Course Notes).** $\widehat{=}$ Slides + Comments



$\rightsquigarrow$ yellow parts in table of contents (left) already covered in lectures.

FAU Michael Kohlhase: SMAI 13 2025-05-06

The central idea in the AI4AI approach – using AI to support learning AI – and thus the ALeA system is that we want to make course materials – i.e. what we give to students for preparing and postparing lectures – more like teachers and study groups (only available 24/7) than like static books.

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

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

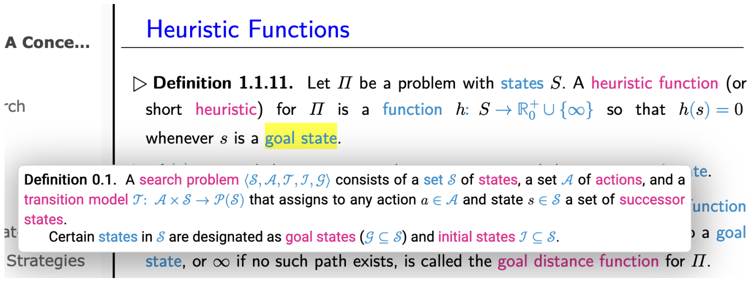
- **SMAI in ALeA:** `https://courses.voll-ki.fau.de/course-home/smai`
  - All details for the course.
  - recorded syllabus (keep track of material covered in course)
  - syllabus of the last semesters (for over/preview)
- **ALeA Status:** The ALeA system is deployed at FAU for over 1000 students taking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

FAU Michael Kohlhase: SMAI 14 2025-05-06

The ALeA SMAI page is the central entry point for working with the ALeA system. You can get to all the components of the system, including two presentations of the course contents (notes- and slides-centric ones), the flashcards, the localized forum, and the quiz dashboard.
We now come to the heart of the ALeA system: its learning support services, which we will now briefly introduce. Note that this presentation is not really sufficient to undertstand what you may be getting out of them, you will have to try them, and interact with them sufficiently that the learner model can get a good estimate of your competencies to adapt the results to you.

Learning Support Services in ALeA

- **Idea:** Embed learning support services into active course materials.
- **Example 1.3.3 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)



A Conce…

ch

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.

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.

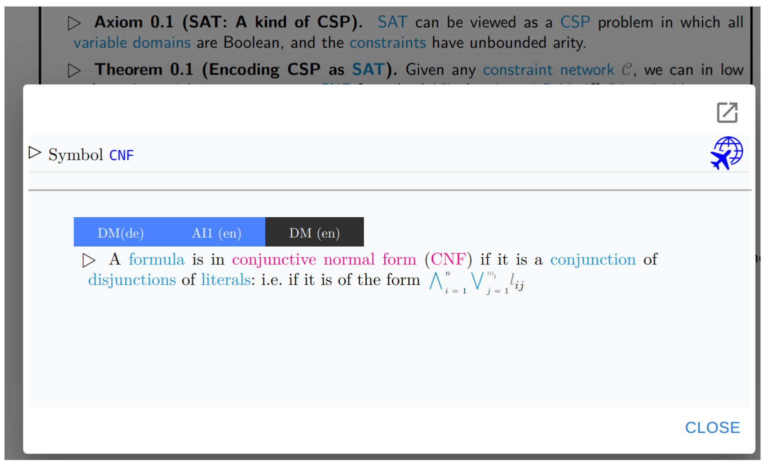
Certain states in $\mathcal{S}$ are designated as goal states ($\mathcal{G} \subseteq \mathcal{S}$) and initial states $\mathcal{I} \subseteq \mathcal{S}$.

te. unction a goal

Strategies

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

▷ **Example 1.3.4 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.



▷ **Axiom 0.1 (SAT: A kind of CSP).** SAT can be viewed as a CSP problem in which all variable domains are Boolean, and the constraints have unbounded arity.

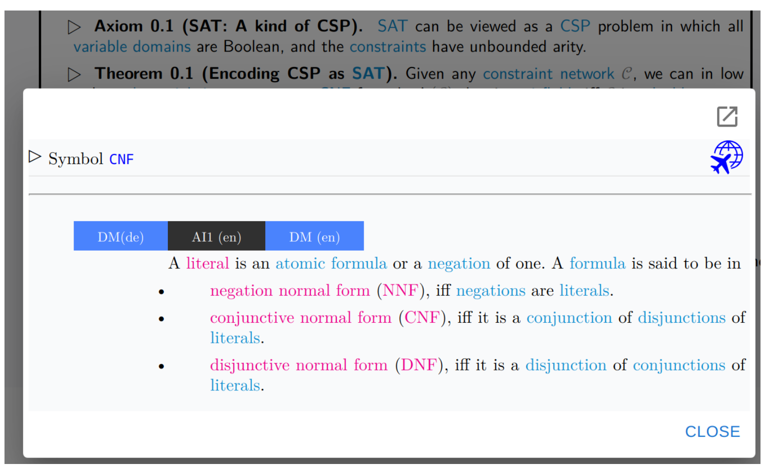
▷ **Theorem 0.1 (Encoding CSP as SAT).** Given any constraint network $\mathcal{C}$, we can in low

▷ Symbol CNF

DM(de) AI1 (en) DM (en)

▷ A formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals: i.e. if it is of the form $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{ij}$

CLOSE



▷ **Axiom 0.1 (SAT: A kind of CSP).** SAT can be viewed as a CSP problem in which all variable domains are Boolean, and the constraints have unbounded arity.

▷ **Theorem 0.1 (Encoding CSP as SAT).** Given any constraint network $\mathcal{C}$, we can in low

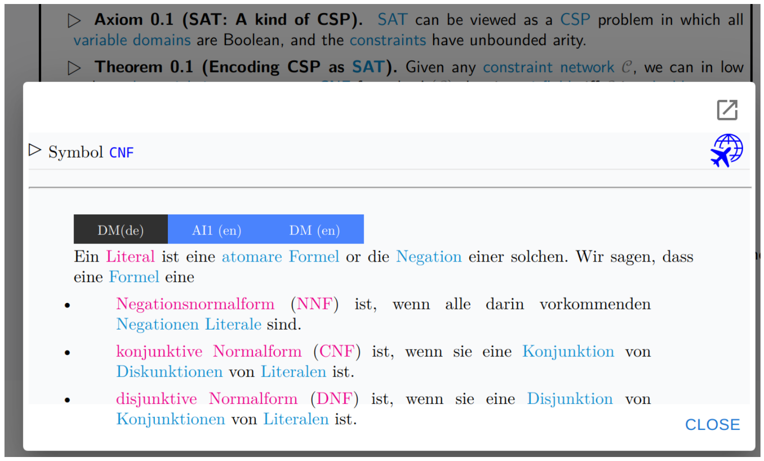
▷ Symbol CNF

DM(de) AI1 (en) DM (en)

A literal is an atomic formula or a negation of one. A formula is said to be in

- negation normal form (NNF), iff negations are literals.
- conjunctive normal form (CNF), iff it is a conjunction of disjunctions of literals.
- disjunctive normal form (DNF), iff it is a disjunction of conjunctions of literals.

CLOSE



▷ **Example 1.3.5 (Guided Tour).** A guided tour for a concept c assembles definitions/etc. into a self-cont

c = countable $\leadsto$

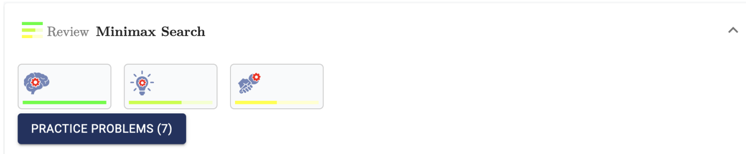
▷ . . . your idea here . . . (the sky is the limit)

Note that this is only an initial collection of learning support services, we are constantly working on additional ones. Look out for feature notifications () on the upper right hand of the ALeA screen.

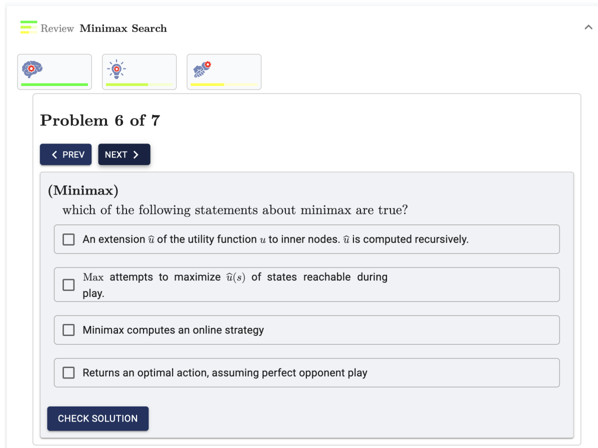
(Practice/Remedial) Problems Everywhere

- ▷ **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)



While the learning support services up to now have been adressed to individual learners, we now turn to services addressed to communities of learners, ranging from study groups with three learners, to whole courses, and even – eventually – all the alumni of a course, if they have not de-registered from ALeA.

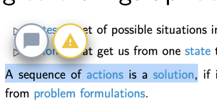
Currently, the community aspect of ALeA only consists in localized interactions with the course materials.

The ALeA system uses the semantic structure of the course materials to localize some interactions that are otherwise often from separate applications. Here we see two:

1. one for reporting content errors – and thus making the material better for all learners – and‘‘
2. a localized course forum, where forum threads can be attached to learning objects.

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.)

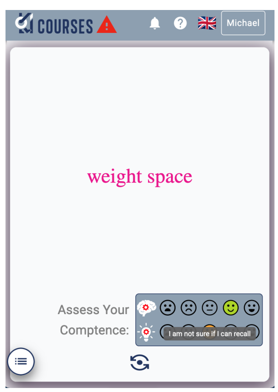
▷ Answering questions gives karma $\widehat{=}$ a public measure of user helpfulness.

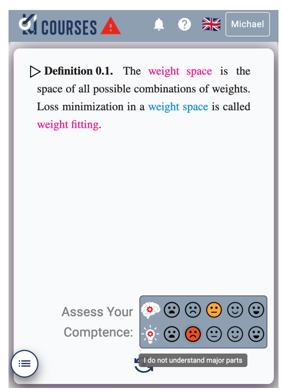
▷ Notes can be anonymous (⇝ generate no karma)

We can use the same four models discussed in the space of guided tours to deploy additional learning support services, which we now discuss.

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)

We have already seen above how the learner model can drive the drilling with flashcards. It can also be used for the configuration of card stacks by configuring a domain e.g. a section in the course materials and a competency threshold. We now come to a very important issue that we always face when we do AI systems that interface with humans. Most web technology companies that take one the approach "the user pays for the services with their personal data, which is sold on" or integrate advertising for renumeration. Both are not acceptable in university setting.

But abstaining from monetizing personal data still leaves the problem how to protect it from intentional or accidental misuse. Even though the GDPR has quite extensive exceptions for research, the ALeA system – a research prototype – adheres to the principles and mandates of the GDPR. In particular it makes sure that personal data of the learners is only used in learning support services directly or indirectly initiated by the learners themselves.

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.

Michael Kohlhase: SMAI 19 2025-05-06

So, now that you have an overview over what the ALEA system can do for you, let us see what you have to concretely do to be able to use it.

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)

FAU Michael Kohlhase: SMAI 20 2025-05-06

Even if you did not understand some of the AI jargon or the underlying methods (yet), you should be good to go for using the ALEA system in your day-to-day work.

Chapter 2

Foundations: Mathematical Language in Practice

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

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

Let's start with the math!

Discrete Math for the moment

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

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

2.1 Mathematical Foundations: Natural Numbers

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

Something very basic:

- ▷ Numbers are symbolic representations of numeric quantities.
- ▷ There are many ways to represent numbers (more on this later)
- ▷ let's take the simplest one (about 8,000 to 10,000 years old)

- ▷ we count by making marks on some surface.
- ▷ For instance //// stands for the number four (be it in 4 apples, or 4 worms)
- ▷ Let us look at the way we construct numbers a little more algorithmically,
- ▷ **Definition 2.1.1.** 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 2.1.2.** For ////, apply the o-rule once and then the s-rule four times.
- ▷ **Definition 2.1.3.** we call these representations unary natural numbers.

FAU Michael Kohlhase: SMAI 22 2025-05-06

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

Instead of using bones, the slash / is a more convenient symbol, but it is manipulated in the same way as in the most ancient times of mankind. The o-rule allows us to start with a blank slate or an empty container like a bowl. The s- or successor-rule allows to put an additional bone into a bowl with bones, respectively, to append a slash to a sequence of slashes. For instance //// stands for the number four — be it 4 apples, or 4 worms. This representation is constructed by applying the o-rule once and then the s-rule four times. So, we have a basic understanding of natural numbers now, but we will also need to be able to talk about them in a mathematically precise way. Generally, this additional precision will involve defining specialized vocabulary for the concepts and objects we want to talk about, making the assumptions we have about the objects exmplicit, and the use of special modes or argumentation.

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

With the notion of a successor from Definition 2.1.4 we can formulate a set of assumptions (called axioms) about unary natural numbers. We will want to use these assumptions (statements we believe to be true) to derive other statements, which — as they have been obtained by generally accepted argumentation patterns — we will also believe true. This intuition is put more formally in Definition 2.1.6 below, which also supplies us with names for different types of statements.

A little more sophistication (math) please

- ▷ **Definition 2.1.4.** 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 2.1.5.** /// is the successor of // and // the predecessor of ///.
- ▷ **Definition 2.1.6.** The following set of axioms are called the Peano axioms (Giuseppe Peano $*1858$, $\dagger 1932$)
- ▷ **Axiom 2.1.7 (P1).** *" " (aka. "zero") is a unary natural number. " " (aka. "zero") is a unary natural number.*
- ▷ **Axiom 2.1.8 (P2).** *Every unary natural number has a successor that is a unary natural number and that is different from it.*
- ▷ **Axiom 2.1.9 (P3).** *Zero is not a successor of any unary natural number.*
- ▷ **Axiom 2.1.10 (P4).** *Different unary natural numbers have different successors.*
- ▷ **Axiom 2.1.11 (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?)

Note that the Peano axioms may not be the first things that come to mind when thinking about characteristic properties of natural numbers. Indeed they have been selected to to be minimal, so that we can get by with as few assumptions as possible; all other statements of properties can be derived from them, so minimality is not a bug, but a feature: the Peano axioms form the foundation, on which all knowledge about unary natural numbers rests.

2.2 Reasoning about Natural Numbers

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

Reasoning about Natural Numbers

- ▷ The Peano axioms can be used to reason about natural numbers.
- ▷ **Definition 2.2.1.** An axiom (or postulate) is a statement about mathematical objects that we assume to be true.
- ▷ **Definition 2.2.2.** 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 2.2.3.** We call a sequence of inferences a derivation or a proof (of the last statement).

FAU Michael Kohlhase: SMAI 24 2025-05-06

If we want to be more precise about these (important) notions, we can define them as follows:
Definition 2.2.4. In general, a axiom is a starting point in logical reasoning with the aim to prove a mathematical statement (called a conjecture as long as it is unproven and unrefuted). A conjecture that is proven is called a theorem.

Conventionally, there are there are two subtypes of theorems. A lemma is an intermediate theorem that serves as part of a proof of a larger theorem. A corollary is a theorem that follows directly from another theorem.

A logical system consists of axioms and rules that allow inference, i.e. that allow to form new formal statements out of already proven ones. So, a proof of a conjecture starts from the axioms that are transformed via the rules of inference until the conjecture is derived.

We will now practice this reasoning on a couple of examples. Note that we also use them to introduce the inference system of mathematics via these example proofs.
Here are some theorems you may want to prove for practice. The proofs are relatively simple.

Let's practice derivations and proofs

- ▷ **Example 2.2.5.** //////////// is a unary natural number
- ▷ **Theorem 2.2.6.** */// is a different unary natural number than //.*
- ▷ **Theorem 2.2.7.** *///// is a different unary natural number than //.*
- ▷ **Theorem 2.2.8.** *There is a unary natural number of which /// is the successor*
- ▷ **Theorem 2.2.9.** *There are at least 7 unary natural numbers.*

▷ **Theorem 2.2.10.** *Every unary natural number is either zero or the successor of a unary natural number.* (we will come back to this later)

Induction for unary natural numbers

▷ **Theorem 2.2.11.** *Every unary natural number is either zero or the successor of a unary natural number.*

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

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

□

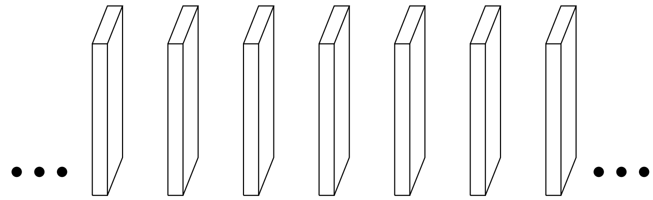
We have already seen in the proof above, that it helps to give names to objects: for instance, by using the name n for the number about which we assumed the property P, we could just say that $P(n)$ holds. But there is more we can do.
Theorem 2.2.11 is a very useful fact to know, it tells us something about the form of unary natural numbers, which lets us streamline induction proofs and bring them more into the form you may know from school: to show that some property P holds for every natural number, we analyze an arbitrary number n by its form in two cases, either it is zero (the base case), or it is a successor of another number (the step case). In the first case we prove the base case and in the latter, we prove the step case and use the induction axiom to conclude that all natural numbers have property P. We will show the form of this proof in the domino-induction below.

The Domino Theorem

▷ **Theorem 2.2.12.** *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.



Michael Kohlhase: SMAI 27 2025-05-06

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".

□

Michael Kohlhase: SMAI 28 2025-05-06

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

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

2.3 Defining Operations on Natural Numbers

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

What can we do with unary natural numbers?

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

▷ **Definition 2.3.1 (The addition "function").** We "define" the addition operation $\oplus$ procedurally (by an algorithm)

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

▷ **Questions:** to understand this definition, we have to know

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

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

Definition 2.3.2 (Replacement). If we have a representation s of an object and we have an equation $l = r$, then we can obtain an object by replacing an occurrence of the sub-expression l in s by r and have $s = s'$.

In other words if we replace a sub-expression of s with an equal one, nothing changes. This is exactly what we will use the two defining equations from Definition 2.3.1 for in the following example:

Example 2.3.3 (Computing the Sum Two and One). If we start with the expression $s(s(o)) \oplus s(o)$, then we can use the second equation to obtain $s(s(s(o)) \oplus o)$ (replacing equals by equals), and – this time with the first equation $s(s(s(o)))$.

Observe: In the computation in Example 2.3.3 at every step there was exactly one of the two equations we could apply. This is a consequence of the fact that in the second argument of the two equations are of the form o and $s(n)$: by Theorem 2.2.11 these two cases cover all possible natural numbers and by **P**3 (see ???), the equations are mutually exclusive. As a consequence we do not really have a choice in the computation, so the two equations do form an "algorithm" (to the extend we already understand them), and the operation is indeed well-defined.

Definition 2.3.4. The form of the arguments in the two equations in Definition 2.3.1 is the same as in the induction axiom, therefore we will consider the first equation as the base equation and second one as the step equation.

We can understand the process of computation as a "getting-rid" of operations in the expression. Note that even though the step equation does not really reduce the number of occurrences of the operator (the base equation does), but it reduces the number of constructor in the second argument, essentially preparing the elimination of the operator via the base equation. Note that in any case when we have eliminated the operator, we are left with an expression that is completely made up of constructors; a representation of a unary natural number. Now we want to see whether we can find out some properties of the addition operation. The method for this is of course stating a conjecture and then proving it.

Addition on unary natural numbers is associative

▷ **Theorem 2.3.5.** *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 $s(n \oplus m \oplus l) = n \oplus m \oplus s(l)$

□

We observe that In the proof above, the induction corresponds to the defining equations of $\oplus$; in particular base equation of $\oplus$ was used in the base case of the induction whereas the step equation of $\oplus$ was used in the step case. Indeed computation (with operations over the unary natural numbers) and structural induction (over unary natural numbers) are just two sides of the same coin as we will see. Let us consider a couple more operations on the unary natural numbers to fortify our intutions.

More Operations on Unary Natural Numbers

- ▷ **Definition 2.3.6.** 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 2.3.7.** 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 2.3.8.** 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 2.3.9.** The unary product operation can be defined by the equations $\bigodot_{i=o}^{o}(n_i) = s(o)$ and $\bigodot_{i=o}^{s(m)}(n_i) = n_{s(m)} \odot \bigodot_{i=o}^{m}(n_i)$.

In Definition 2.3.6, we have used the operation $\oplus$ in the right-hand side of the step-equation. This is perfectly reasonable and only means that we have eliminate more than one operator.
Note that we did not use disambiguating parentheses on the right hand side of the step equation for $\odot$. Here $n \oplus n \odot m$ is a unary sum whose second summand is a product. Just as we did there, we will use the usual arithmetic precedences to reduce the notational overload.
The remaining examples are similar in spirit, but a bit more involved, since they nest more operators. Just like we showed associativity for $\oplus$ in slide 30, we could show properties for these operations, e.g.

$$\bigodot_{i=o}^{(n \oplus m)}(k_i) = \bigodot_{i=o}^{n}(k_i) \odot \bigodot_{i=o}^{m}(k_{i \oplus n}) \tag{2.1}$$

by induction, with exactly the same observations about the parallelism between computation and induction proofs as $\oplus$.
Definition 2.3.9 gives us the chance to elaborate on the process of definitions some more: When we define new operations such as the product over a sequence of unary natural numbers, we do have freedom of what to do with the corner cases, and for the "empty product" (the base case equation) we could have chosen

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

But any value but $s(o)$ would violate the generalized distributivity law in equation 2.1 which is exactly what we would expect to see (and which is very useful in calculations). So if we want to have this equation (and I claim that we do) then we have to choose the value $s(o)$.

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

Chapter 3

Talking (and Writing) about Mathematics

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

Talking about Mathematics

- **Definition 3.0.1.** 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)

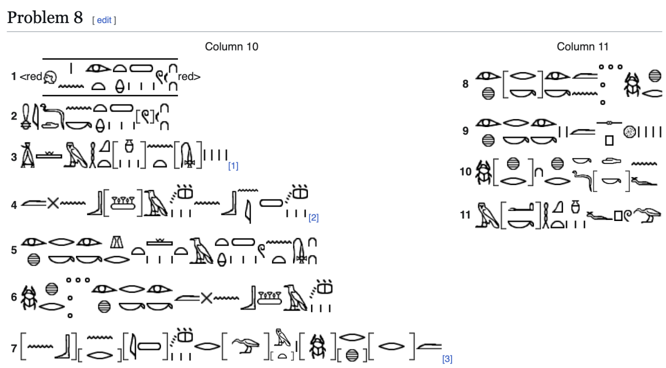
  We call this language mathematical vernacular.
- **Definition 3.0.2.** A technical language is a natural language extended by a terminology and (possibly) special idioms, discourse markers, and notations.
- **Definition 3.0.3.** A jargon (or terminology) is a set of specialized words or phrases (called 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 3.1.1 (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)
- ▷ 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".
- ▷ 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 3.1.2 (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.

▷ **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 3.1.3.** 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 3.1.4.** Mathematical vernacular aggregates objects/statements for cognitive efficiency.

- ▹ "*$a \in S$, $b \in S$, and $c \in S$ ⇝ "$a, b, c \in S$"*," (object aggregation)
- ▹ "*$i \geq 0$ and $i \leq n$*" ⇝ "$0 < i < n$", (statement aggregation)
- ▹ "*For all n with $n > 0$ ...*" ⇝ "*For all $n > 0$ ...*" (apposition; note seeming grammatical conflict)

▷ **Definition 3.1.5.** 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 3.1.6.** Mathematical vernacular uses ellipses (...) as a constructor for sequences.

The meaning of an ellipsis is usually considered "obvious" and left for interpretation by the reader.

▷ **Example 3.1.7.** 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 3.1.8.**

- ▹ 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.

Definition 3.2.1. 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 3.2.2.** Abbreviations for mathematical statements in MathTalk

- ▷ $\wedge$ and $\vee$ are common notations for "*and*" and "*or*"
- ▷ "*not*" is in mathematical statements often denoted with $\neg$
- ▷ $\forall x.P$ ($\forall x{\in}S.P$) stands for "*condition P holds for all x (in S)*"
- ▷ $\exists x.P$ ($\exists x{\in}S.P$) stands for "*there exists an x (in S) such that proposition P holds*"
- ▷ $\nexists x.P$ ($\nexists x{\in}S.P$) stands for "*there exists no x (in S) such that proposition P holds*"
- ▷ $\exists^1 x.P$ ($\exists^1 x{\in}S.P$) stands for "*there exists one and only one x (in S) such that proposition P holds*"
- ▷ iff as abbreviation for "*if and only if*", symbolized by "$\Leftrightarrow$"
- ▷ the symbol "$\Rightarrow$" is used a as shortcut for "*implies*"; we can read $A \Rightarrow B$ as "*if A then B*".

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

▷ **Example 3.2.3.** $\forall x.\exists y.x = y \Leftrightarrow \neg x \neq y$ reads

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

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

Peano Axioms in MathTalk

▷ **Example 3.2.4.** We can write the Peano Axioms in MathTalk: If we write $n \in \mathbb{N}_1$ for "*n is a unary natural number*", and $P(n)$ for "*n has property P*", then we can write

- ▷ $o \in \mathbb{N}_1$ (zero is a unary natural number)
- ▷ $\forall n{\in}\mathbb{N}_1.s(n) \in \mathbb{N}_1 \wedge n \neq s(n)$ ($\mathbb{N}_1$ closed under successors, distinct)
- ▷ $\neg(\exists n{\in}\mathbb{N}_1.o = s(n))$ (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 3.2.5.** We often see clauses like "*Let* $\epsilon, \delta > 0 \ldots$", 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 3.2.6.** 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 3.2.7.** 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* $\epsilon, \delta > 0, \ldots$"
  - but declarations are grammatically clauses, so the sentence structure becomes simpler. (especially when iterated)

We will use mathematical vernacular throughout the remainder of the SMAI notes. The abbreviations will mostly be used in informal communication situations. Many mathematicians consider it bad style to use abbreviations in printed text, but approve of them as parts of formulae (see e.g. Definition 4.1.3 for an example).

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

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 3.2.8.** 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.

We will now discuss four definition schemata – specific well-understood forms a definition can take – that will occur over and over in this course.

Definition Schemata – Simple/Pattern Definition

▷ **Definition 3.2.9.** 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 3.2.10.** 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 3.2.11.** 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 3.2.12.** 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 3.2.13.** An implicit definition (also called definition by description) is an clause or expression $\mathbf{A}$, such that we can prove “there is exactly one n such that $\mathbf{A}$" ($\exists^1 n.\mathbf{A}$), where n is a new name – the definiendum.

If such an unique existence proof exists, we call n well-defined.

▷ **Example 3.2.14.** $\forall x.x \in \emptyset$ 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 3.2.15.** 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.

Definition 3.2.16. 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 3.2.17.** 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 $\delta := \epsilon$ ..." 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, ...
- ...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 3.3.1.** 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 3.3.2.** *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 3.3.3.** 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 3.3.4.** 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 3.3.5 (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 3.3.6.** 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 3.3.7.** 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 3.3.8.** 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 3.3.9.** 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 3.3.10.** 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 $x \rightsquigarrow$ 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.)

▷ **Definition 3.3.11.** 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 3.3.12.** *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.

▷ **Example 3.3.13.** 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 3.3.14.** 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 3.3.15.** 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 3.3.16.** 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 3.3.17.** 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 3.3.18 (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 3.3.19.** 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 3.3.20.** 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 3.3.21 (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".
- **Claim:** There is a system behind this, which makes math communication very efficient.
- **Definition 3.3.22.** 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)

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

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!

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

Chapter 4

Elementary Discrete Math

4.1 Naive Set Theory

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

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

Understanding Sets

- Sets are one of the foundations of mathematics, ...
- ... and one of the most difficult concepts to get right axiomatically.
- **Early Definition Attempt:** A set is "everything that can form a unity in the face of God". (Georg Cantor ($*1845$, $\dagger 1918$))
- 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 4.1.1 (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 4.1.2.** *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)

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

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

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

Relations between Sets

- ▷ **Definition 4.1.3.** set equality: $(A \equiv B):\equiv(\forall x.x \in A \Leftrightarrow x \in B)$
- ▷ **Definition 4.1.4.** subset: $(A \subseteq B):\equiv(\forall x.x \in A \Rightarrow x \in B)$
- ▷ **Definition 4.1.5.** proper subset: $(A \subset B):\equiv(A \subseteq B) \wedge (A \not\equiv B)$
- ▷ **Definition 4.1.6.** superset: $(A \supseteq B):\equiv(\forall x.x \in B \Rightarrow x \in A)$
- ▷ **Definition 4.1.7.** proper superset: $(A \supset B):\equiv(A \supseteq B) \wedge (A \not\equiv B)$

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

Operations on Sets

- ▷ **Definition 4.1.8.** union: $A \cup B := \{x \mid x \in A \vee x \in B\}$
- ▷ **Definition 4.1.9.** 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 4.1.10.** intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$
- ▷ **Definition 4.1.11.** 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 4.1.12.** set difference: $A \backslash B := \{x \mid x \in A \wedge x \notin B\}$
- ▷ **Definition 4.1.13.** the power set: $\mathcal{P}(A) := \{S \mid S \subseteq A\}$
- ▷ **Definition 4.1.14.** the empty set: $\forall x.x \notin \emptyset$
- ▷ **Definition 4.1.15.** Cartesian product: $A \times B := \{(a,b) \mid a \in A \wedge b \in B\}$, call (a,b) pair.
- ▷ **Definition 4.1.16.** 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 4.1.17.** 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 4.1.18.** 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 $\bigcap_{i \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\}} S_i$.

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

Sizes of Sets

- We would like to talk about the size of a set. Let us try a definition
- **Definition 4.1.19.** 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)

FAU Michael Kohlhase: SMAI 64 2025-05-06

Once we try to prove the identifies from Definition 64 we get into problems. Even though the notion of "counting the elements of a set" is intuitively clear (indeed we have been using that since we were kids), we do not have a mathematical way of talking about associating numbers with objects in a way that avoids double counting and skipping. We will have to postpone the discussion of sizes until we do. But before we delve in to the notion of relations and functions that we need to associate set members and counting let us now look at large sets, and see where this gets us.

Sets can be Mind-boggling

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

FAU Michael Kohlhase: SMAI 65 2025-05-06

A less scary $\mathcal{S}$?

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

  In either case, we have $\mathcal{S}' \in \mathcal{S}'$ iff $\mathcal{S}' \notin \mathcal{S}'$, which is a contradiction! (Russell's Antinomy [Bertrand Russell '03])
- **Does MathTalk help?:** no: $\mathcal{S}' := \{m \mid m \notin m\}$
  - MathTalk allows statements that lead to contradictions, but are legal wrt. "vocabulary" and "grammar".
- We have to be more careful when constructing sets! (axiomatic set theory)
- **for now:** stay away from large sets. (stay naive)

Even though we have seen that naive set theory is inconsistent, we will use it for this course. But we will take care to stay away from the kind of large sets that we needed to construct the paradox. Now we will take a closer look at two very fundamental notions in mathematics that can be built on the notion of sets introduced above: relations and functions. We have already encountered functions and relations as set operations — e.g. the elementhood relation $\in$ which relates a set to its elements or the power set function that takes a set and produces another (its power set).

4.2 Relations

Intuitively, relations are mathematical objects that take arguments and state whether they are related in a partiular way.

Relations

- **Definition 4.2.1.** $R \subseteq A \times B$ is a (binary) relation between A and B.
- **Definition 4.2.2.** If $A = B$ then R is called a relation on A.
- **Definition 4.2.3.** $R \subseteq A \times B$ is called total iff $\forall x \in A. \exists y \in B. (x,y) \in R$.
- **Definition 4.2.4.** $R^{-1} := \{(y,x) \mid (x,y) \in R\}$ is the converse relation of R.
- **Note:** $R^{-1} \subseteq B \times A$.
- **Definition 4.2.5.** 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 4.2.6.**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.

We will need certain classes of relations in following, so we introduce the necessary abstract properties of relations. We will later combine these to obtain types of relations that behave like well-known ones like the "*less than*", "*less-or-equal*", or the equality relation.

Properties of binary Relations

- **Definition 4.2.7 (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 4.2.8.** The equality relation is an equivalence relation on any set.
- **Example 4.2.9.** On sets of persons, the "mother-of" relation is an non-symmetric, non-reflexive relation.

Indeed the equivalence relation defined last is a generalization of the equality relation – which is symmetric, reflexive, and transitive. We will see later that any equivalence relation behaves a bit like equality: we can do the same things with it. That makes the class of equivalence relations useful and interesting.

The abstract properties defined above allow us to easily define another very important class of relations, the ordering relations, which generalize the well-known "*less than*" and "*less than or equal*" relations: We just combine some other elementary properties.

Strict and Non-Strict Partial Orders

- **Definition 4.2.10.** 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 4.2.11 (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 4.2.12.** The $\leq$ relation is a linear order on $\mathbb{N}$ (all elements are comparable)

▷ **Example 4.2.13.** The "ancestor-of" relation is a partial order that is not linear.

▷ **Lemma 4.2.14.** *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 4.2.15.** *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

Intuitively, functions are mathematical objects that take arguments (as input) and return a result (as output). This already suggests defining them as special relations. But we have to be careful here; we want to specify the sets where the inputs can come from (the domain) and where the outputs go (the codomain), and whether a function needs to have outputs for all possible inputs (from the domain). This leads us to a distinction of "total" and "partial" functions.

Functions (as special relations)

▷ **Definition 4.3.1.** $f \subseteq X \times Y$, 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 4.3.2.** call X the domain (write $\mathbf{dom}(f)$), and Y the codomain ($\mathbf{codom}(f)$) (come with f)

▷ **Notation:** $f(x) = y$ instead of $(x,y) \in f$ (function application)

▷ **Definition 4.3.3.** 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 4.3.4.** 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 4.3.5.** 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!)

As functions are foundational in mathematics, we see a lot of suggestive notations, but they should not hide the fact that functions are just "right unique" relations. Definition 4.3.1 gives us a solid foundation on which we can reason safely about functions.

Remark: It is crucial to understand that the domain and codomain is part of a functions (partial or total). In particular, a change in domain or codomain changes the function.

Example 4.3.6. The functions $f\colon \mathbb{R}\to\mathbb{R}; x\mapsto|x|$, $g\colon \mathbb{R}^+\to\mathbb{R}; x\mapsto|x|$, and $h\colon \mathbb{R}^+\to\mathbb{R}^+; x\mapsto|x|$ are different – they have different domains. In particular have different properties as we will see later.

Now that we have defined functions, it is natural to think about sets of functions from a given domain to a given codomain. If the latter are small enough, we can even write down the full set as a collection of sets of pairs.

Function Spaces

- **Definition 4.3.7.** 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 4.3.8.** 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)

We will now introduce still another notation for functions, which is commonly used in CS, since it is more explicit in the arguments a function takes and allows to construct functions without directly having to give them a name.

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 $\mathbf{dom}(f)$. But if we do not want to use a name, we can only say $\mathbf{dom}(\{(x,y)\in\mathbb{R}\times\mathbb{R}\,|\,y=x^2+3x+5\})$
- **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 4.3.9.** 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 4.3.10.** The simplest function we always try everything on is the identity function:

$$\lambda n\in\mathbb{N}.n = \{(n,n)\,|\,n\in\mathbb{N}\} = \mathrm{Id}_{\mathbb{N}} = \{(0,0),(1,1),(2,2),(3,3),\ldots\}$$

▷ **Example 4.3.11.** We can also to more complex expressions, here we take the square function

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

Lambda-Notation for Functions (continued)

▷ **Example 4.3.12.** λ notation also works for more complicated domains. In this case we have pairs as arguments.

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

The three properties we define next give us information about whether we can invert functions, i.e. whether the converse relation of a given function is again a (partial) function.

Properties of functions, and their converses

▷ **Definition 4.3.13.** 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 4.3.14.** *If f is injective, then the converse relation f^{-1} is a partial function.*

▷ **Observation 4.3.15.** *If f is surjective, then the converse f^{-1} is a total relation.*

▷ **Definition 4.3.16.** If f is bijective, call the converse relation inverse function, we (also) write it as f^{-1}.

▷ **Observation 4.3.17.** *If f is bijective, then f^{-1} is a total function.*

▷ **Observation 4.3.18.** *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 4.3.19.** 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

With the notion of bijectivity defined above, we can make progress on the notion of the "size" of a set we failed on Definition 4.1.19.

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 4.3.20.** 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 4.3.21.** 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 4.3.22.** *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 $\rightsquigarrow$ Homework)

Michael Kohlhase: SMAI 76 2025-05-06

Next we turn to operations on functions. These are actually functions themselves, they take functions as arguments, and may return functions as results. We call such functions higher-order. For instance the composition function takes two functions as arguments and yields a function as a result.

Operations on Functions

- **Definition 4.3.23.** 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 4.3.24.** 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 4.3.25.** Let $f\colon A \to B$ be a function, $A' \subseteq A$ and $B' \subseteq B$, then we call
  - $f(A') := \{b \in B \mid \exists a {\in} A'.(a,b) \in f\}$ the image of A' under f,
  - $\mathbf{Im}(f) := f(A)$ the image of f, and
  - $f^{-1}(B') := \{a \in A \mid \exists b {\in} B'.(a,b) \in f\}$ the preimage of B' under f

Michael Kohlhase: SMAI 77 2025-05-06

4.4 Equivalence Relations and Quotients

Equivalence and Equality

- ▷ **Recap:** We have defined equivalence relation as a reflexive, symmetric, and transitive relation.
- ▷ **Example 4.4.1.** Equality is an equivalence relation. (trivially)
- ▷ **Example 4.4.2.** 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 4.4.3 (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 4.4.4.** 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 4.4.5.** *Equality is the most fine-grained (i.e. smallest wrt. the partial ordering $\subseteq$) equivalence relation.* *(it distinguishes most)*
- ▷ **Lemma 4.4.6.** *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 4.4.7.** Let S be a set and R be an equivalence relation on S, then for any $x \in S$ we call the set $[x]_R := \{y \in S \mid R(x, y)\}$ the equivalence class of x (under R), and the set $S/R := \{[x]_R \mid x \in S\}$ 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 4.4.8.** 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 4.4.9.** 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 4.4.10.** Remember: $n \equiv_k m$, iff $k|(n-m)$. It is an equivalence relation and $\pi_{\equiv_k} : \mathbb{Z}/\equiv_k \to \{n \mid 0 \leq n \leq (k-1)\}$ is bijective.
- ▷ **Lemma 4.4.11.** *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

We will use Standard ML (SML) as the primary programming language for the course. This has three reasons:

- The mathematical foundations of the computational model of SML is very simple: it consists of functions, which we have already studied. You will be exposed to imperative programming languages (`C` and `C++`) in the labs and later in your studies.
- As a functional programming language, SML introduces two very important concepts in a very clean way: typing and recursion.
- Finally, SML has a very useful secondary virtue for a course in our program, where students come from very different backgrounds: it provides a (relatively) level playing ground, since it is unfamiliar to all students.

Enough theory, let us start computing with functions

- ▷ We will use Standard ML (SML) in this course.
- ▷ **Definition 5.1.1.** 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".

FAU Michael Kohlhase: SMAI 80 2025-05-06

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

Historical Remark: The name ML comes from the phrase "Meta Language": ML was developed as the scripting language for a tactical theorem prover[1] — a program that can construct

[1] The "Edinburgh LCF" system

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

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

Standard ML (SML)

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

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

- A book by Bob Harper (CMU) `http://www-2.cs.cmu.edu/~rwh/smlbook/`
- The Moscow ML home page, one of the ML's that you can try to install, it also has many interesting links `https://mosml.org/`.
- The home page of SML-NJ (SML of New Jersey), the most commonly used SML implementation `http://www.smlnj.org/` also has a ML interpreter and links Online Books, Tutorials, Links, FAQ, etc. And of course you can download SML from there for Unix as well as for Windows.
- and finally a page on ML by the people who originally invented ML: `http://www.lfcs.inf.ed.ac.uk/software/ML/`.

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

```
— ⟨⟨input to the interpreter⟩⟩
⟨⟨system response⟩⟩
```

Programming in SML (Basic Language)

- **Generally:** Start the SML interpreter, play with the program state.
- **Definition 5.1.2 (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 ∗)

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

Definition 5.1.3. types are program constructs that classify program objects into categories.

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

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

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

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

Programming in SML (Declarations)

- **Definition 5.1.4.** 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 5.1.5.** A local declaration uses **let** to bind variables in its scope (delineated by **in** and **end**).
- **Example 5.1.6.** 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
```


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

Programming in SML (Component Selection)

▷ **Definition 5.1.7.** Using structured patterns, we can declare more than one variable. We call this pattern matching.

▷ **Example 5.1.8 (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 5.1.9.** Anonymous variables (if we are not interested in one value)

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

▷ **Example 5.1.10.** We can define the selector function for pairs in SML as

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

▷ Note the type: SML supports universal types with type variables 'a, 'b,. . . .
▷ first is a function that takes a pair of type 'a∗'b as input and gives an object of type 'a as output.

Another unusual but convenient feature realized in SML is the use of pattern matching. In pattern matching we allow to use variables (previously unused identifiers) in declarations with the understanding that the interpreter will bind them to the (unique) values that make the declaration true.

In our example the second input contains the variables x and y. Since we have bound the identifier unitvector to the value (1,1), the only way to stay consistent with the state of the interpreter is to bind both x and y to the value 1.

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

What's next?

More SML constructs and general theory of functional programming.

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

Using SML lists

- SML has a built-in "list type" (actually a list type constructor)
- Given a type ty, list ty is also a type.

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

- Constructors nil and :: (nil $\widehat{=}$ 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?)

Definition 5.1.11. A constructor is an operator that "constructs" members of an SML data type. The type of lists has two constructors: nil that "constructs" a representation of the empty list, and the "list constructor" :: (we pronounce this as "cons"), which constructs a new list h::l from a list l by pre-pending an element h (which becomes the new head of the list).

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

Definition 5.1.12. We say that the type of lists is inductive.

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

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 5.1.13.** We write $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots \rightsquigarrow t_n$ for tracing the recursive arguments t_i through a recursive computation.

▷ **Example 5.1.14.** We have the following evaluation trace with result [2,3,4]

upto(2,4) ⇝ 2::upto(3,4) ⇝ 2::(3::upto(4,4)) ⇝ 2::(3::(4::nil))

▷ **Definition 5.1.15.** We call an SML function recursive, iff the function is called in the function definition.

▷ **Example 5.1.16.** Note that recursive functions need not terminate, consider the function

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

which has the evaluation sequence

diverges(1) ⇝ 1 + diverges(2) ⇝ 1 + (2 + diverges(3)) ⇝ ...

The key to understanding recursion is to understand the function as an equation – as the SML syntax already suggests – that replaces any expression that matches the left hand side with a suitably instantiated version of the right hand side. Using this, we can trace the computation (we write ⇝ for any such replacement).
Note that not all computations stop with a base case: we may have forgotten to specify one, or or computation never reaches it. This is actually a feature of recursion, not a bug. The full power of programming languages necessarily comes with the "ability" to obtain infinite computations. In imperative languages, we call these infinite loops, in functional programming languages like SML, we speak of "deep recursions".
Normally of course, we want our recursive computations to terminate after a finite number which can be large of steps. For that to happen, something needs to become smaller in the computation. In Example 5.1.14, this is the difference between the two arguments, it decreases by one in each step of the computation, and thus finally reaches zero, where the first alternative of the **if** expression applies. In the function of Example 5.1.16, the argument does not get smaller, indeed it becomes bigger with every recursive call, leading to the divergent behavior.
In our examples above we used recursion on an argument of type int using if_then_else expression to select between the base case and the step case. recursion on list types is more elegant, since we can use pattern matching on the arguments.

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 5.1.17.** 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"]
```


To understand pattern matching in 88, consider the type string list list – for the argument here, we only need the fact that we are dealing with a list type here. And in those, elements are either nil or of the form cons(h,t) – i.e. a non-empty list made by pre-pending an element h to a list t. With the pattern matching mechanism we can select them directly, e.g.

```
- val h::t = [1,2];
val h = 1 : int
val t = [2] : int list
- val h::t = [1];
val h = 1 : int
val t = [] : int list
```

This is just what we do in the flat function. We have two cases, in the second – the step case – we match the argument (which is non-empty, since the first case already took care of the empty list) with the pattern h::t, which binds the parameters h and t, which we can then use to construct the value of flattening a non-empty list.
Defining functions by cases and recursion is a very important programming mechanism in SML. At the moment we have only seen it for the built-in type of lists. In the future we will see that it can also be used for user-defined data types.
We will now look at the string type of SML and how to deal with it. But before we do, let us recap what strings are.
Definition 5.1.18. Strings are just sequences of characters.
Therefore, SML just provides an interface to lists for manipulation.

Lists and Strings

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

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

- ▷ **Exercise:** Try to come up with a function that detects palindromes like 'otto' or 'anna', try also (more at [Pal])
  - ▷ 'Marge lets Norah see Sharon's telegram', or (up to case, punct and space)
  - ▷ 'Erika feuert nur untreue Fakire' (for German speakers)

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

Higher-Order Functions

- **Idea:** Pass functions as arguments (functions are normal values.)
- **Example 5.1.19.** 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 5.1.20.** We can program the map function ourselves!

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

- **Example 5.1.21.** Declaring functions (yes, functions are normal values.)

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

- **Example 5.1.22.** 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;
```

- **Definition 5.1.23.** We call functions that take functions as arguments higher-order functions and those that do not first-order functions.

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

Cartesian and Cascaded Functions

- We have not been able to treat binary, ternary,... functions directly
- **Workaround 1:** Make use of (Cartesian) products. (unary functions on tuples)
- **Example 5.1.24.** $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ with $+(\langle 3,2 \rangle)$ instead of $+(3,2)$

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

- **Workaround 2:** Make use of functions as results.
- **Example 5.1.25.** $+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ withn $+(3)(2)$ instead of $+(\langle 3,2 \rangle)$.

```
- 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.

FAU Michael Kohlhase: SMAI 91 2025-05-06

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

Cartesian and Cascaded Functions (Brackets)

- **Definition 5.1.26.** Call a function Cartesian, iff the argument type is a product type, call it cascaded, iff the result type is a function type.
- **Example 5.1.27.** 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
  - $e_1\ e_2\ e_3 \rightsquigarrow (e_1\ e_2)\ e_3$ (function application associates to the left)
  - $\tau_1{->}\tau_2{->}\tau_3 \rightsquigarrow \tau_1\ {->}(\tau_2{->}\tau_3)$ (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
```

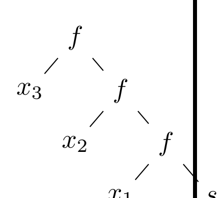
FAU Michael Kohlhase: SMAI 92 2025-05-06

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

Folding Operators

- **Definition 5.1.28.** SML provides the left folding operator to realize a recurrent computation schema

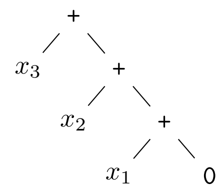
```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl f s [x1,x2,x3] = f(x3,f(x2,f(x1,s)))
```



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

▷ **Example 5.1.29.** Folding the iterator **op**+ with start value 0:

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



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

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

Note: We have used the binary function **op**+ as the argument to the foldl operator instead of simply + in Example 5.1.29. The reason for this is that the infix notation for x +y is syntactic sugar for **op**+(x,y) and not for +(x,y) as one might think.

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

Finally note: foldl is a cascaded function. SML could have made it Cartesian, resulting in the type ('a * 'b –> 'b) * 'b * 'a list –> 'b. But the cascaded foldl is more useful as you we do things like

```
- val sum = foldl op+ 0;
val sum = fn : int list -> int
```

Note that we are leaving over the list argument to get a function. If SML used Cartesian, then we would have to define the equivalent

```
- fun sum (l) = foldl op+ 0 l
val sum = fn : int list -> int;
```

which is longer. Also we can pass summation as an argument more elegantly as in

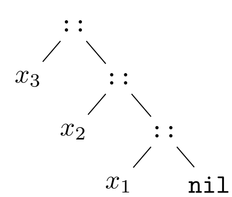
```
map (foldl op+ 0) [[1,2,3],[2,3,4],[67]]
```

with a cascaded foldl.

Folding Procedures (continued)

▷ **Example 5.1.30 (Reversing Lists).**

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



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

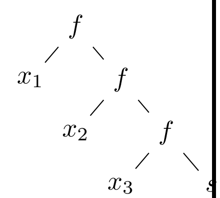
In Example 5.1.30, we reverse a list by folding the list constructor (which duly constructs the reversed list in the process) over the input list; here the empty list is the right start value.

Folding Procedures (foldr)

▷ **Definition 5.1.31.** The right folding operator foldr is a variant of foldl that processes the list elements in reverse order.

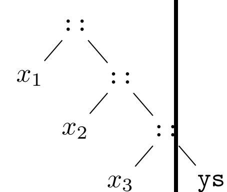
```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

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



▷ **Example 5.1.32 (Appending Lists).**

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



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

Michael Kohlhase: SMAI 95 2025-05-06

In Example 5.1.32 we fold with the list constructor again, but as we are using foldr the list is not reversed. To get the append operation, we use the list in the second argument as a base case of the iteration.

Now that we know some SML

SML is a functional programming language

What does this all have to do with functions?

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

Michael Kohlhase: SMAI 96 2025-05-06

5.2 Inductively Defined Sets and Computation

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

What about Addition, is that a function?

▷ **Problem:** Addition takes two arguments (binary function)

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

▷ **Definition 5.2.1 (Defining equations).** $+((n,o)) = n$ (base) and $+((m,s(n))) = s(+((m,n)))$ (step)

▷ **Theorem 5.2.2.** $+ \subseteq \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$ *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

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

Addition is a total Function

▷ **Lemma 5.2.3.** *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 $l' = l$.

□

▷ **Corollary 5.2.4.** $+: \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$ *is a total function.*

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

Reflection: How could we do this?

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

▷ **Observation:** Defining Equations for $+$: $+((n,o)) = n$ (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 5.2.5.** Defining equations $\delta(o) = o$ and $\delta(s(n)) = s(s(\delta(n)))$

▷ **Example 5.2.6.** 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

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

As we have identified the necessary conditions for proving function-hood, we can now generalize the situation, where we can obtain functions via defining equations: we need inductively defined sets, i.e. sets with Peano-like axioms. This observation directly leads us to a very important concept in computing.

Inductively Defined Sets

▷ **Definition 5.2.7.** An inductively defined set $\langle S, C\rangle$ is a set S together with a finite set $C := \{c_i \,|\, 1 \leq i \leq n\}$ of k_i ary constructors $c_i : S^{k_i} \to S$ 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)
- $\mathbf{Im}(c_i) \cap \mathbf{Im}(c_j) = \emptyset$ for $i \neq j$, and (no confusion between constructors)
- for every $s \in S$ there is a constructor $c \in C$ with $s \in \mathbf{Im}(c)$. (no junk)

▷ Note that we also allow nullary constructors here.

▷ **Example 5.2.8.** $\langle \mathbb{N}_1, \{s, o\}\rangle$ is an inductively defined set.

▷ *Proof:* We check the three conditions in Definition 5.2.7 using the Peano Axioms

1. Generation is guaranteed by $\mathbf{P}1$ and $\mathbf{P}2$
2. Internal confusion is prevented $\mathbf{P}4$
3. Inter-constructor confusion is averted by $\mathbf{P}3$
4. Junk is prohibited by $\mathbf{P}5$.

□

This proof shows that the Peano axiom are exactly what we need to establish that $\langle\mathbb{N}_1, \{s, o\}\rangle$ is an inductively defined set. Now that we have invested so much elbow grease into specifying the concept of an inductively defined set, it is natural to ask whether there are more examples. We will look at a particularly important one next.

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

- **Lists of (unary) natural numbers:** $[1, 2, 3]$, $[7, 7]$, $[]$, ...
  - nil-rule: start with the empty list $[]$
  - cons-rule: extend the list by adding a number $n \in \mathbb{N}_1$ at the front
- **Definition 5.2.9.** 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 5.2.10.** e.g. $[3, 2, 1] \mathrel{\widehat{=}} \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, \mathrm{nil})))$ and $[] \mathrel{\widehat{=}} \mathrm{nil}$
- **Definition 5.2.11.** 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 Definition 2.1.6.
- **Axiom 5.2.12 (LP1).** $\mathrm{nil} \in \mathcal{L}[\mathbb{N}]$ *(generation axiom (nil))*
- **Axiom 5.2.13 (LP2).** $\mathrm{cons}\colon \mathbb{N}_1 \times \mathcal{L}[\mathbb{N}] \to \mathcal{L}[\mathbb{N}]$ *(generation axiom (cons))*
- **Axiom 5.2.14 (LP3).** *nil is not a cons-value*
- **Axiom 5.2.15 (LP4).** *cons is injective*
- **Axiom 5.2.16 (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.

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

Note furthermore that the $\mathbf{P}i$ and the $\mathbf{LP}i$ are very similar in structure: they say the same things about the constructors.

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

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

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

A direct consequence of this observation is that

Corollary 5.2.17. *The set $\langle\mathbb{N}_1 \cup \mathcal{L}[\mathbb{N}], \{o, s, \mathrm{nil}, \mathrm{cons}\}\rangle$ is an inductively defined set in the sense of Definition 5.2.7.*

Operations on Lists: Append

▷ **Definition 5.2.18.** 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 5.2.19.** $[3,2,1]@[1,2] = [3,2,1,1,2]$ and $[]@[1,2,3] = [1,2,3] = [1,2,3]@[]$

▷ **Lemma 5.2.20.** *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 5.2.21.** *Append is a function* *(see, this just worked fine!)*

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

Operations on Lists: more examples

▷ **Definition 5.2.22.** $\lambda(\mathrm{nil}) = o$ and $\lambda(\mathrm{cons}(n,l)) = s(\lambda(l))$

▷ **Definition 5.2.23.** $\rho(\mathrm{nil}) = \mathrm{nil}$ and $\rho(\mathrm{cons}(n,l)) = \rho(l)@\mathrm{cons}(n,\mathrm{nil})$.

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

5.3 Inductively Defined Sets in SML

We are about to introduce one of the most powerful aspects of SML, its ability to let the user define types. After all, we have claimed that types in SML are first-class objects, so we have to have a means of constructing them.
We have seen above, that the main feature of an inductively defined set is that it has Peano Axioms that enable us to use it for computation. Note that specifying them, we only need to know the constructors (and their types). Therefore the **datatype** constructor in SML only needs to specify this information as well. Moreover, note that if we have a set of constructors of an inductively defined set — e.g. `zero : mynat` and `suc : mynat −> mynat` for the set `mynat`, then their codomain type is always the same: `mynat`. Therefore, we can condense the syntax even further by leaving that implicit.

Data Type Declarations

- **Definition 5.3.1.** SML data types provide concrete syntax for inductively defined sets via the keyword **datatype** followed by a list of constructor declarations.
- **Example 5.3.2.** 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 5.3.3.** *We can define functions by (complete) case analysis over the constructors*
- **Example 5.3.4 (Converting types).**

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

- **Example 5.3.5 (Missing Constructor Cases).**

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

- **Example 5.3.6 (Inconsistency).**

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


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

Data Types Example (Enumeration Type)

- **Example 5.3.7.** A type for weekdays (nullary constructors)

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

- **Example 5.3.8.** 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

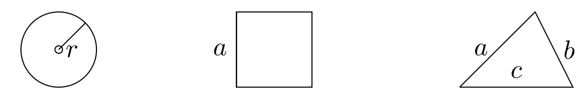
```
- weekend sun
true : bool
- map weekend [mon, wed, fri, sat, sun]
[false, false, false, true, true] : bool list
```

- Nullary constructors describe values, enumeration types finite sets.

Somewhat surprisingly, finite enumeration types that are separate constructs in most programming languages are a special case of **datatype** declarations in SML. They are modeled by sets of base constructors, without any functional ones, so the base cases form the finite possibilities in this type. Note that if we imagine the Peano Axioms for this set, then they become very simple; in particular, the induction axiom does not have step cases, and just specifies that the property P has to hold on all base cases to hold for all members of the type.
Let us now come to a real-world examples for data types in SML. Say we want to supply a library for talking about mathematical shapes (circles, squares, and triangles for starters), then we can represent them as a data type, where the constructors conform to the three basic shapes they are in. So a circle of radius r would be represented as the constructor term Circle r (what else).

Data Types Example (Geometric Shapes)

▷ Describe three kinds of geometrical forms as mathematical objects



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

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

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

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

▷ This gives us the constructor functions

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


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

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

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

Data Types Example (Areas of Shapes)

▷ **Example 5.3.9.** 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
```


All three kinds of shapes are included in one abstract entity: the type shape, which makes programs like the area function conceptually simple — it is just a function from type shape to type real. The complexity — after all, we are employing three different formulae for computing the area of the respective shapes — is hidden in the function body, but is nicely compartmentalized, since the constructor cases in systematically correspond to the three kinds of shapes.

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

Chapter 6

Graphs and Trees

We will first introduce the formal definitions of graphs (trees will turn out to be special graphs), and then fortify our intuition using some examples.

Basic Definitions: Graphs

- **Definition 6.0.1.** 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 6.0.2.** 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 6.0.3.** 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.

FAU Michael Kohlhase: SMAI 109 2025-05-06

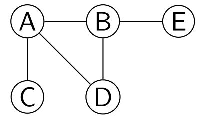
We will mostly concentrate on directed graphs in the following, since they are most important for the applications we have in mind. Many of the notions can be defined for undirected graphs with a little imagination. For instance the definitions for indeg and outdeg are the obvious variants: $\mathrm{indeg}(v)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

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

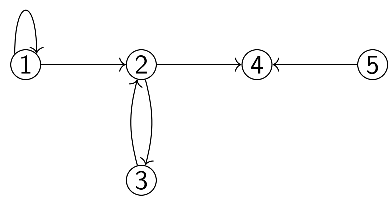
This is a very abstract yet elementary definition. We only need very basic concepts like set and pair to understand them. The main difference between directed and undirected graphs can be visualized in the graphic representations below:

Examples

▷ **Example 6.0.4.** 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 6.0.5.** A directed graph $G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{(1,1), (1,2), (2,3), (3,2), (2,4), (5,4)\}$



In a directed graph, the edge (shown as the connections between the circular node) have a direction (mathematically they are pairs), whereas the edges in an undirected graph do not (mathematically, they are represented as a set of two elements, in which there is no natural order).

Note furthermore that the two diagrams are not graphs in the strict sense: they are only pictures of graphs. This is similar to the famous painting by René Magritte that you have surely seen before.

The Graph Diagrams are not Graphs

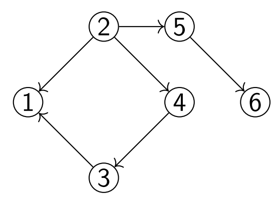
They are pictures of graphs (of course!)

If we think about it for a while, we see that directed graphs are nothing new to us. We have defined a directed graph to be a set of pairs over a base set (of nodes). These objects we have seen in the beginning of this course and called them relations. So directed graphs are special relations. We will now introduce some nomenclature based on this intuition.

Directed Graphs

- **Idea:** Directed graphs are nothing else than relations.
- **Definition 6.0.6.** 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 6.0.7.** The node 2 is initial, and the nodess 1 and 6 are terminal in



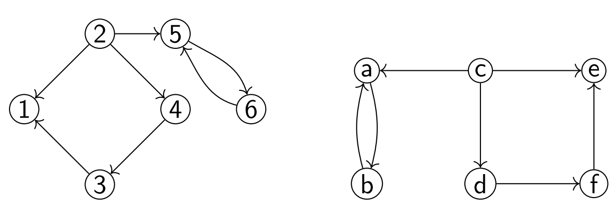
For mathematically defined objects it is always very important to know when two representations are "equal" (i.e. indistinguishable). We have already seen this for sets, where $\{a, b\}$ and $\{b, a, b\}$ represent the same set: the set with the elements a and b. In the case of graphs, the condition is a little more involved: we have to find a bijection of nodes that respects the edges.

Graph Isomorphisms

- **Definition 6.0.8.** 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

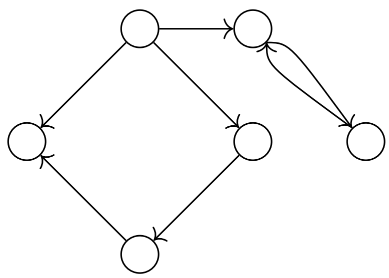
directed graphs	undirected graphs
$(a,b) \in E \Leftrightarrow (\psi(a),\psi(b)) \in E'$	$\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in E'$

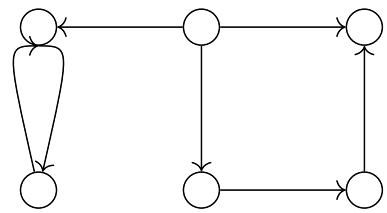
- **Definition 6.0.9.** Two graphs G and G' are equivalent iff there is an isomorphism ψ between G and G'.
- **Example 6.0.10.** G_1 and G_2 are equivalent as there exists a isomorphism $\psi := \{a \mapsto 5, b \mapsto 6, c \mapsto 2, d \mapsto 4, e \mapsto 1, f \mapsto 3\}$ between them.



Note that we have only marked the circular nodes in the diagrams with the names of the elements that represent the nodes for convenience, the only thing that matters for graphs is which nodes are connected to which. Indeed that is just what the definition of graph equivalence via the existence of an isomorphism says: two graphs are equivalent, iff they have the same number of nodes and the same edge connection pattern. The objects that are used to represent them are purely coincidental, they can be changed by an isomorphism at will. Furthermore, as we have seen in the example, the shape of the diagram is purely an artefact of the presentation; It does not matter at all.

So the following two diagrams stand for the same graph, (it is just much more difficult to state the isomorphism)





Note that directed and undirected graphs are totally different mathematical objects. It is easy to think that an undirected edge $\{a, b\}$ is the same as a pair $(a,b), (b,a)$ of directed edges in both directions, but a priory these two have nothing to do with each other. They are certainly not equivalent; we only have equivalence between directed graphs and also between undirected graphs, but not between graphs of differing classes.

Now that we understand graphs, we can add more structure. We do this by defining a labeling function from nodes and edge.

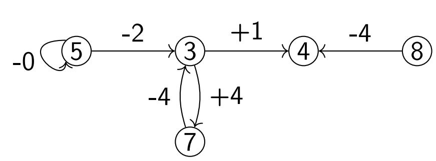
Labeled Graphs

▷ **Definition 6.0.11.** A labeled graph G is a quadruple $\langle V, E, L, l\rangle$ where $\langle V, E\rangle$ is a graph and $l\colon V \cup E \to L$ 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 6.0.12.** $G = \langle V, E, L, l\rangle$ with $V = \{A, B, C, D, E\}$, where

- ▷ $E = \{(A,A), (A,B), (B,C), (C,B), (B,D), (E,D)\}$
- ▷ $l\colon V \cup E \to \{+, -, \emptyset\} \times \{1, \ldots, 9\}$ with
  - ▷ $l(A) = 5$, $l(B) = 3$, $l(C) = 7$, $l(D) = 4$, $l(E) = 8$,
  - ▷ $l((A,A)) = -0$, $l((A,B)) = -2$, $l((B,C)) = +4$,
  - ▷ $l((C,B)) = -4$, $l((B,D)) = +1$, $l((E,D)) = -4$



Note that in this diagram, the markings in the nodes do denote something: this time the labels given by the labeling function l, not the vertices – the objects used to construct the graph. This is somewhat confusing, but traditional.

Now we come to a very important concept for graphs. A path is intuitively a sequence of nodes that can be traversed by following directed edges in the right direction or undirected edges.

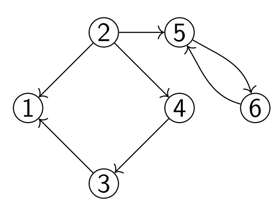
Paths in Graphs

- **Definition 6.0.13.** 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 $(\exists i.v_i = v)$
  - $e \in p$, iff $e = (v,v') \in E$ is an edge on the path $(\exists i.v_i = v \wedge v_{i+1} = v')$
- **Notation:** We write $\Pi(G)$ for the set of all paths in a graph G.

An important special case of a path is one that starts and ends in the same node. We call it a cycle. The problem with cyclic graphs is that they contain paths of infinite length, even if they have only a finite number of nodes.

Cycles in Graphs

- **Definition 6.0.14.** 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 6.0.15.** $(2,4), (4,3)$ 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 6.0.16.** We will sometimes use the abbreviation DAG for "directed acyclic graph".

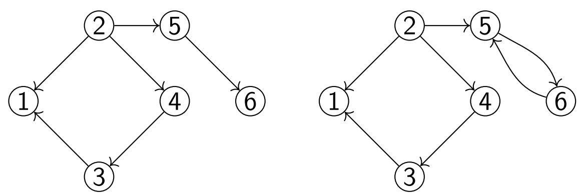
Of course, speaking about cycles is only meaningful in directed graphs, since undirected graphs can only be acyclic, iff they do not have edges at all.

Graph Depth

▷ **Definition 6.0.17.** 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 6.0.18.** 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 6.0.19.** 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)

We now come to a very important special class of graphs, called trees.

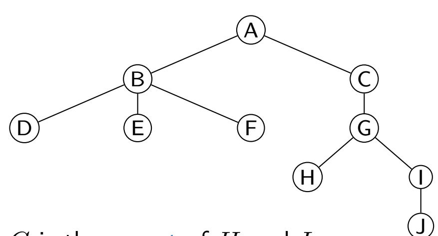
Trees

▷ **Definition 6.0.20.** 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 6.0.21.** 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 6.0.22.** *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*)

In CS, trees are traditionally drawn upside-down with their root at the top, and the leaves at the bottom. The only reason for this is that (like in nature) trees grow from the root upwards and if we draw a tree it is convenient to start at the top of the page downwards, since we do not have to know the height of the picture in advance.

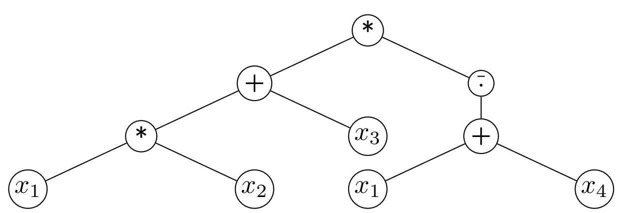
Let us now look at a prominent example of a tree: the parse tree of a Boolean expression. Intuitively, this is the tree given by the brackets in a Boolean expression. Whenever we have an expression of the form $\mathbf{A} \circ bB$, then we make a tree with root $\circ$ and two subtrees, which are constructed from $\mathbf{A}$ and $\mathbf{B}$ in the same manner.

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

The Parse-Tree of a Boolean Expression

- **Definition 6.0.23.** 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 6.0.24.**

  The parse tree of $(x_1 * x_2 + x_3) * \overline{x_1 + x_4}$ is



Chapter 7

Recap: Formal Languages and Grammars

One of the main ways of designing rational agents in this course will be to define formal languages that represent the state of the agent environment and let the agent use various inference techniques to predict effects of its observations and actions to obtain a world model. In this chapter we recap the basics of formal languages and grammars that form the basis of a compositional theory for them.

The Mathematics of Strings

- **Definition 7.0.1.** 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 7.0.2.** 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 "abc" for $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}\rangle$
- **Example 7.0.3.** 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 7.0.4 (String Length).** Given a string s we denote its length with $|s|$.
- **Definition 7.0.5.** The concatenation $\mathrm{conc}(s, t)$ of two strings $s = \langle s_1, ..., s_n\rangle \in A^n$ and $t = \langle t_1, ..., t_m\rangle \in A^m$ is defined as $\langle s_1, ..., s_n, t_1, ..., t_m\rangle \in A^{n+m}$.

  We will often write $\mathrm{conc}(s, t)$ as $s + t$ or simply st
- **Example 7.0.6.** $\mathrm{conc}(\text{"text"}, \text{"book"}) = \text{"text"} + \text{"book"} = \text{"textbook"}$

FAU Michael Kohlhase: SMAI 120 2025-05-06

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

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

Formal Languages

- **Definition 7.0.7.** 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 7.0.8.** If $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then $A^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \ldots, \mathtt{aaa}, \ldots\}$.
- **Definition 7.0.9.** A set $L \subseteq A^*$ is called a formal language over A.
- **Definition 7.0.10.** We use $\mathtt{c}^{[n]}$ for the string that consists of the character $\mathtt{c}$ repeated n times.
- **Example 7.0.11.** $\#^{[5]} = \langle\#, \#, \#, \#, \#\rangle$
- **Example 7.0.12.** The set $M := \{\mathtt{ba}^{[n]} \,|\, n \in \mathbb{N}\}$ of strings that start with character $\mathtt{b}$ followed by an arbitrary numbers of $\mathtt{a}$'s is a formal language over $A = \{\mathtt{a}, \mathtt{b}\}$.
- **Definition 7.0.13.** 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: $L_1L_2 := \{uw \,|\, u \in L_1,\ w \in L_2\}$. We often use L_1L_2 instead of L_1L_2.
  - Language power L: $L^0 := \{\epsilon\}$, $L^{n+1} := LL^n$, where $L^n := \{\mathtt{w}_1 \ldots \mathtt{w}_n \,|\, w_i \in L$, 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: $L^R := \{w^R \,|\, w \in L\}$.

FAU Michael Kohlhase: SMAI 121 2025-05-06

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

Of course this definition only shifts complexity to the way we construct specific formal languages (where it actually belongs), and we have learned two (simple) ways of constructing them: by repetition of characters, and by concatenation of existing languages. As mentioned above, the purpose of a formal language is to distinguish "good" from "bad" strings. It is maximally general, but not helpful, since it does not support computation and inference. In practice we will be interested in formal languages that have some structure, so that we can represent formal languages in a finite manner (recall that a formal language is a subset of A^*, which may be infinite and even undecidable – even though the alphabet A is finite).

To remedy this, we will now introduce phrase structure grammars (or just grammars), the standard tool for describing structured formal languages.

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 7.0.14.** 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 $h \rightarrow b_1 | \ldots | b_n$ instead.

We fortify our intuition about these – admittedly very abstract – constructions by an example and introduce some more vocabulary.

Phrase Structure Grammars (cont.)

▷ **Example 7.0.15.** A simple phrase structure grammar G:

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

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

▷ **Definition 7.0.16.** 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 7.0.17. 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 7.0.18.** The non-lexicon production rules are called structural, and the nonterminals in the heads are called phrasal or syntactic categories.

Now we look at just how a grammar helps in analyzing formal languages. The basic idea is that a grammar accepts a word, iff the start symbol can be rewritten into it using only the rules of the grammar.

Phrase Structure Grammars (Theory)

- **Idea:** Each symbol sequence in a formal language can be analyzed/generated by the grammar.
- **Definition 7.0.19.** 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 $s \rightarrow_G^p t$ (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 7.0.20.** 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 7.0.21.** The language $\mathbf{L}(G)$ of G is the set of its sentences. We say that $\mathbf{L}(G)$ is generated by G.

  Definition 7.0.22. We call two grammars equivalent, iff they have the same languages.

  Definition 7.0.23. A grammar G is said to be universal if $\mathbf{L}(G) = \Sigma^*$.
- **Definition 7.0.24.** 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.

Michael Kohlhase: SMAI 124 2025-05-06

Again, we fortify our intuitions with Example 7.0.15.

Phrase Structure Grammars (Example)

- **Example 7.0.25.** In the grammar G from Example 7.0.15:
  1. $Article$ **teacher** Vi is a sentential form,
  $$\begin{aligned} S &\rightarrow_G NP\ Vi \\ &\rightarrow_G Article\ N\ Vi \\ &\rightarrow_G Article\ \textbf{teacher}\ Vi \end{aligned}$$
  2. "*The teacher sleeps*" is a sentence.
  $$\begin{aligned} S &\rightarrow_G^* Article\ \textbf{teacher}\ Vi \\ &\rightarrow_G \textbf{the teacher}\ Vi \\ &\rightarrow_G \textbf{the teacher sleeps} \end{aligned}$$

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

Michael Kohlhase: SMAI 125 2025-05-06

Note that this process indeed defines a formal language given a grammar, but does not provide an efficient algorithm for parsing, even for the simpler kinds of grammars we introduce below.

Grammar Types (Chomsky Hierarchy [Cho65])

- **Observation:** The shape of the grammar determines the "size" of its language.
- **Definition 7.0.26.** 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.
- **Example 7.0.27 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$ is accepted by

$$\begin{array}{rcl} S & \rightarrow & \mathbf{a}\,\mathbf{b}\,\mathbf{c} \mid A \\ A & \rightarrow & \mathbf{a}\,A\,B\,\mathbf{c} \mid \mathbf{a}\,\mathbf{b}\,\mathbf{c} \\ \mathbf{c}\,B & \rightarrow & B\,\mathbf{c} \\ \mathbf{b}\,B & \rightarrow & \mathbf{b}\,\mathbf{b} \end{array}$$

- **Example 7.0.28 (Context-free).** The language $\{a^{[n]}b^{[n]}\}$ is accepted by $S \rightarrow \mathbf{a}\ S\ \mathbf{b} \mid \epsilon$.
- **Example 7.0.29 (Regular).** The language $\{a^{[n]}\}$ is accepted by $S \rightarrow S\ \mathbf{a}$
- **Observation:** Natural languages are probably context-sensitive but parsable in real time! (like languages low in the hierarchy)

FAU Michael Kohlhase: SMAI 126 2025-05-06

While the presentation of grammars from above is sufficient in theory, in practice the various grammar rules are difficult and inconvenient to write down. Therefore CS – where grammars are important to e.g. specify parts of compilers – has developed extensions – notations that can be expressed in terms of the original grammar rules – that make grammars more readable (and writable) for humans. We introduce an important set now.

Useful Extensions of Phrase Structure Grammars

- **Definition 7.0.30.** 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 $s \le c \le t$ 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 (⇝ 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 $Y \to Y\ s$.

We will now build on the notion of BNF grammar notations and introduce a way of writing down the (short) grammars we need in SMAI that gives us even more of an overview over what is happening.

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	$\mathbf{A}$	$::=$	$p^k(t_1, \ldots, t_k)$	atomic
		$\mid$	$\neg\mathbf{A}$	negation
		$\mid$	$\mathbf{A}_1 \wedge \mathbf{A}_2$	conjunction
		$\mid$	$\forall X.\mathbf{A}$	quantifier

We will generally get by with context-free grammars, which have highly efficient into parsing algorithms, for the formal language we use in this course, but we will not cover the algorithms in SMAI.

Chapter 8

Term Languages and Abstract Grammars

Term Languages

- ▷ In most applications of symbolic AI, the formal languages are very structured.
- ▷ **Example 8.0.1 (Arithmetic Expressions).** Consider the grammar G and the G-derivation

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

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

  G accepts the string −(((X29+3)∗555)) but not −(X29∗3(.
- ▷ **Definition 8.0.2.** We will call such languages term languages.
- ▷ **Observation:** Strings in $\mathbf{L}(G)$ are "well-bracketed" and can be split into sub-strings from $\mathbf{L}(G)$.
- ▷ **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 8.0.3.** 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 8.0.4.** 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:

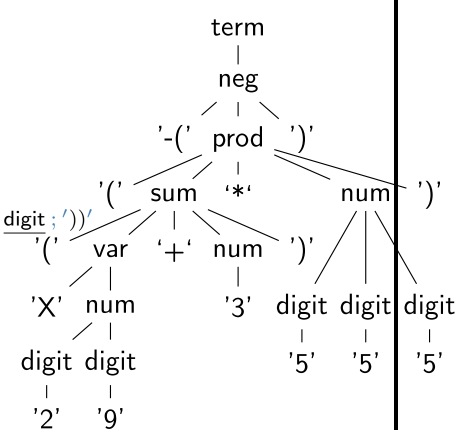
- ▹ If $D = s{\rightarrow^p_G}t$, then p is a lexical rule $s \rightarrow t$ and t consists of a single terminal. Then P_D is the tree whose root has label s with one child labeled with t.
- ▹ If $D = s \rightarrow^p_G s' \rightarrow^*_G t$, $D' := s'{\rightarrow^*}_G t$, and $p = h \rightarrow a_1 \ldots a_n$, 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".

A Parse Tree for −(((X29+3)∗555))

▷ **Example 8.0.5.** A parse tree for (the derivation on the left for) the string −(((X29+3)∗555))

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



▷ 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 8.0.6.** 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 8.0.7 (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 $\leadsto$ "−(((X29+3)∗(X29+3)))"

Computation via Tree Traversal

▷ We can also do more complex tasks via tree traversal:

▷ **Example 8.0.8 (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 Example 8.0.6 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 8.0.9.** 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 8.0.10.** 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 Example 8.0.6 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) $\to$ x

const(c) $\to$ c

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

bind($\beta[v_1, \ldots, v_l]t_1, \ldots, t_k$) $\longrightarrow$ β (b ($v_1 \cdots v_n$), $t_1 \cdots t_n$)

- ▷

Chapter 9

Mathematical Language Recap

We already clarified above that we will use mathematical language as the main vehicle for specifying the concepts underlying the AI algorithms in this course.

In this chapter, we will recap (or introduce if necessary) an important conceptual practice of modern mathematics: the use of mathematical structures.

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 9.0.1.** 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 $h \rightarrow b$.

Note that the idea of mathematical structures has been picked up by most programming languages in various ways and you should therefore be quite familiar with it once you realize the parallelism.

Mathematical Structures in Programming

- **Observation:** Most programming languages have some way of creating "named structures". Referencing components is usually done via "dot notation".
- **Example 9.0.2 (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 9.0.3 (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.

Even if the idea of mathematical structures may be familiar from programming, it may be quite intimidating to some students in the mathematical notation we will use in this course. Therefore will – when we get around to it – use a special overview notation in SMAI. We introduce it below.

In SMAI we use a mixture between Math and Programming Styles

- In SMAI we use mathematical notation, ...
- **Definition 9.0.4.** A structure signature combines the components, their "types", and accessor names of a mathematical structure in a tabular overview.
- **Example 9.0.5.**

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

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

  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?

We now come to an important topic which is not really part of artificial intelligence but which adds an important layer of understanding to this enterprise: We (still) live in the era of Moore's law (the computing power available on a single CPU doubles roughly every two years) leading to an exponential increase. A similar rule holds for main memory and disk storage capacities. And the production of computer (using CPUs and memory) is (still) very rapidly growing as well; giving mankind as a whole, institutions, and individual exponentially grow of computational resources.

In public discussion, this development is often cited as the reason why (strong) AI is inevitable. But the argument is fallacious if all the algorithms we have are of very high complexity (i.e. at least exponential in either time or space). So, to judge the state of play in artificial intelligence, we have to know the complexity of our algorithms.

In this chapter, we will give a very brief recap of some aspects of elementary complexity theory and make a case of why this is a generally important for computer scientists.

To get a feeling what we mean by "fast algorithm", we do some preliminary computations.

Performance and Scaling

- ▷ Suppose we have three algorithms to choose from. (which one to select)
- ▷ Systematic analysis reveals performance characteristics.
- ▷ **Example 10.0.1.** 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	...	...	...

The last number in the rightmost column may surprise you. Does the run time really grow that fast? Yes, as a quick calculation shows; and it becomes much worse, as we will see.

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 10.0.2.** 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	$1.6min$	$2.5mon$	–

So it does make a difference for larger computational problems what algorithm we choose. Considerations like the one we have shown above are very important when judging an algorithm. These evaluations go by the name of "complexity theory".
Let us now recapitulate some notions of elementary complexity theory: we are interested in the worst-case growth of the resources (time and space) required by an algorithm in terms of the sizes of its arguments. Mathematically we look at the functions from input size to resource size and classify them into "big-O" classes, abstracting from constant factors (which depend on the machine thealgorithm runs on and which we cannot control) and initial (algorithm startup) factors.

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in SMAI.
- **Definition 10.0.3.** 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 10.0.4.** 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 10.0.5 (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 10.0.6 (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

- ▷ $\mathcal{O}(4n^3 + 3n + 7^{1000n}) = \mathcal{O}(2^n)$
- ▷ $\mathcal{O}(n) \subset \mathcal{O}(n \cdot \log_2(n)) \subset \mathcal{O}(n^2)$

OK, that was the theory, . . . but how do we use that in practice?
What I mean by this is that given an algorithm, we have to determine the time complexity.
This is by no means a trivial enterprise, but we can do it by analyzing the algorithm instruction by instruction as shown below.

Determining the Time/Space Complexity of Algorithms

▷ **Definition 10.0.7.** 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
  If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
- ▷ variable: need the complexity of the value
  If $\alpha = v$ with $v \in \mathbf{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.

- ▷ application: compose the complexities of the function and the argument
  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
  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
  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
  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
  If α is **while**γ**do**φ**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 $\rightsquigarrow$ recurrences and Master's theorem.

As instructions in imperative programs can introduce new variables, which have their own time complexity, we have to carry them around via the introduced context, which has to be defined co-recursively with the time complexity. This makes Definition 10.0.7 rather complex. The main two cases to note here are

- the variable case, which "uses" the context Γ and
- the assignment case, which extends the introduced context by the time complexity of the value.

The other cases just pass around the given context and the introduced context systematically.
Let us now put one motivation for knowing about complexity theory into the perspective of the job market; here the job as a scientist.

Please excuse the chemistry pictures, public imagery for CS is really just quite boring, this is what people think of when they say "scientist". So, imagine that instead of a chemist in a lab, it's me sitting in front of a computer.

Why Complexity Analysis? (General)

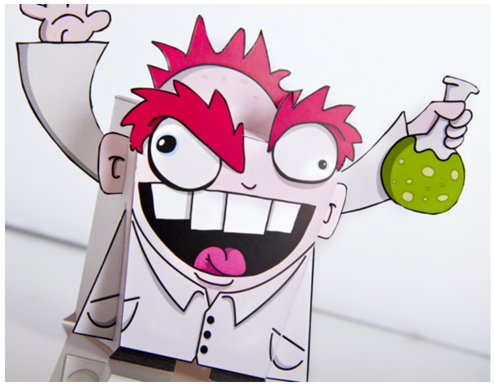
▷ **Example 10.0.8.** Once upon a time I was trying to invent an efficient algorithm.

- ▷ My first algorithm attempt didn't work, so I had to try harder.

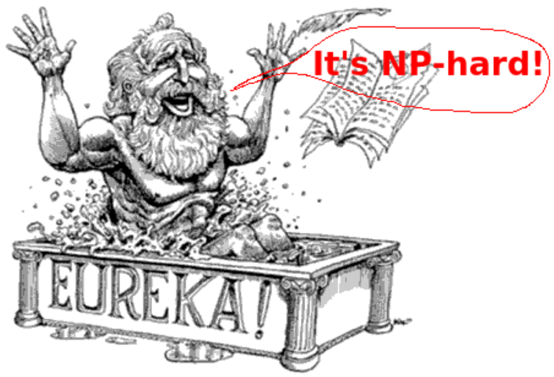
▷ But my 2nd attempt didn't work either, which got me a bit agitated.

▷ The 3rd attempt didn't work either...

▷ And neither the 4th. But then:



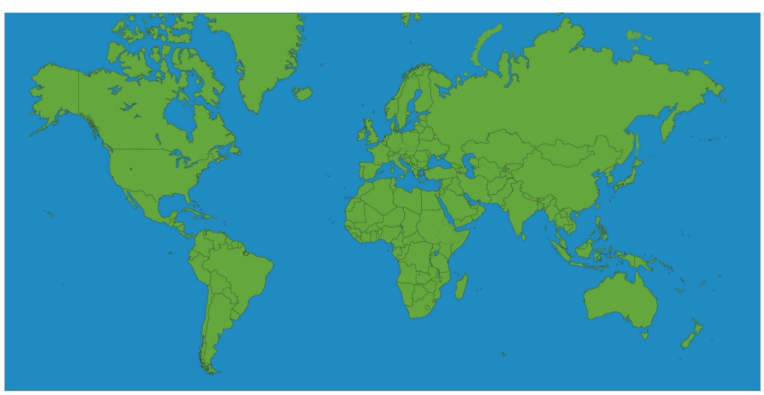
▷ Ta-da ... when, for once, I turned around and looked in the other direction–CAN one actually solve this efficiently? – **NP**-hard was there to rescue me.



The meat of the story is that there is no profit in trying to invent an algorithm, which we could have known that cannot exist. Here is another image that may be familiar to you.

Why Complexity Analysis? (General)

▷ **Example 10.0.9.** 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.

It's like, you're trying to find a route to India (from Spain), and you presume it's somewhere to the east, and then you hit a coast, but no; try again, but no; try again, but no; ... if you don't have a map, that's the best you can do. But the concept "**NP**-hard" gives you the map: you can check that there actually is no way through here. But what is this notion "**NP**-hard" alluded to above? We observe that we can analyze the complexity of problems by the complexity of the algorithms that solve them. This gives us a notion of what to expect from solutions to a given problem class, and thus whether efficient (i.e. polynomial time) algorithms can exist at all.

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 $\mathbf{PSPACE} = \mathbf{NPSPACE}$, and hence (trivially) $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

  It is commonly believed that $\mathbf{NP} \not\supseteq \mathbf{PSPACE}$. (similar to $\mathbf{P} \subseteq \mathbf{NP}$)

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?
- **Answer:** reserved for the plenary sessions $\rightsquigarrow$ be there!

Bibliography

[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.

[Koh08] Michael Kohlhase. "Using LaTeX as a Semantic Markup Format". In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: `https://kwarc.info/kohlhase/papers/mcs08-stex.pdf`.

[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/`.

[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/`.

[sTeX] *sTeX: A semantic Extension of TeX/LaTeX.* URL: `https://github.com/sLaTeX/sTeX` (visited on 05/11/2020).