TU Clausthal
Clausthal University of Technology

Complexity Theory

Prof. Dr. Jürgen Dix

Time and Date: Lecture (SR 210):
Wednesday/Thursday, 10ct.
Exercises/Labs (SR210): approx. fortnightly
Assistent: Niklas Fiekas

Homepage

http://cig.in.tu-clausthal.de/

Visit regularly!

Important info about scripts, slides, exercises et cetera.

Exam: **oral examination.**

What is this lecture about? (1)

This lecture course builds on **“Informatics III”** and deepens many topics introduced there.

The first chapter (5 lectures, 2 labs) starts with Makanin's problem (word equations over a free monoid) which is decidable, and then covers some topics from the Chomsky hierarchy: the Myhill-Nerode theorem (how to construct a minimal finite automaton), deterministic PDA's, type 1 languages, normalform for type 0 and type 1 languages, Ehrenfeucht's theorem and Lindenmayer systems, a class of grammars that is orthogonal to parts of the Chomsky hierarchy.

Chapter 2 (6 lectures, 2 labs) discusses undecidability results in greater depth (Post's Correspondence theorem, (primitive) recursive functions, Hilbert's 10. problem (solving diophantine equations)). We also cover Rice's theorem and its analogue for grammars.

What is this lecture about? (2)

Chapters 3 (2 lectures, 1 lab) and 4 (3 lectures, 1 lab) cover classical complexity results: time versus space, speed-up, gap-theorem, union-theorem and complexity classes: PH (the polynomial hierarchy), PSPACE, EXPSPACE and their subclasses. We also prove Immerman/Szelepcsényi's theorem. We end with the decidability of Presburger arithmetic, the theory of reals with addition (and multiplication) and their respective complexities. The results are obtained by quantifier elimination.

Chapter 5 (1 lecture) is a short overview of the arithmetical and analytical hierarchy (extending EXPSPACE into the undecidable) and discusses descriptive complexity: Can we find logics corresponding to complexity classes (Fagin's theorem).

Many thanks to Nils Bulling for proofreading and helping with the exercises of previous versions.

Table of contents

1. The Chomsky Hierarchy
2. (Un-) Decidability
3. SPACE versus TIME
4. EXPSPACE
5. More advanced topics

1. The Chomsky Hierarchy

1 The Chomsky Hierarchy
- Makanin’s Algorithm
- Minimal DFA
- cf = hom(Dyck$\cap$reg)
- DPDA/DCF
- LBA/Type 1
- re = hom(cs)
- Ehrenfeucht
- Lindenmayer

Content of this chapter:

Type 3:

1. Does a unique minimal automaton exist?
2. PDAs are indeterministic. Which class do deterministic PDA's generate? $\hookrightarrow$ DCF.

Type 2:

1. Dyck = CFL: Each cf-language has the form hom(Dyck $\cap$ type3).

Type 1:

1. **Languages of type 1**: LBA acceptable.
2. Contextsensitive = bounded.
3. Type 1 is closed under complements.

Type 0: Each r.e. set has the form hom(type1).

Content of this chapter:

Ehrenfeucht: Equality of morphisms modulo L can be checked on a finite test set.

New class: **Incomparable to Type 2, 3:**

- Lindenmayer systems: a class which is incomparable to both type 2 and type 3.

A **(homo-) morphism** is a structure-preserving mapping from one algebraic structure in another one.

Definition 1.1 (Homomorphism)

A (homo-) morphism $h : \mathcal{A}_1 \to \mathcal{A}_2$

- of one algebraic structure $\mathcal{A}_1 = \big(M, Op^1_{m_1}, \ldots, Op^k_{m_k}\big)$
- into another algebraic structure of the same type $\mathcal{A}_2 = \big(N, \overline{Op}^1_{m_1}, \ldots, \overline{Op}^k_{m_k}\big)$

is a function $h : M \to N$ that is compatible with all operators. This means that the following holds: for $1 \leqslant i \leqslant k$ and for all $a_1, \ldots, a_{n_i} \in M$:

$$h\big(Op^i_{n_i}(a_1, \ldots, a_{n_i})\big) = \overline{Op}^i_{n_i}\big(h(a_1), \ldots, h(a_{n_i})\big).$$

Definition 1.2 (Isomorphism)

An Isomorphism from $\mathcal{A}_1 = (M, Op^1_{m_1}, \ldots, Op^k_{m_k})$ to $\mathcal{A}_2 = (N, \overline{Op}^1_{m_1}, \ldots, \overline{Op}^k_{m_k})$ is a homomorphism $h : \mathcal{A}_1 \to \mathcal{A}_2$, which is **bijective**.

Example: Graphs.

For languages: a (homo-) morphism from a language based on alphabet Σ to a language based on alphabet Γ, is a function which maps

- one symbol in Σ
- to a word in Γ^* (Γ is a different alphabet).

Some examples on $\leadsto$ **blackboard 1.1**.

Why are morphisms useful?
$\leadsto$ **blackboard 1.2**

Lemma 1.3 (Unique Extension to Σ^*)

h can be uniquely extended *to $h : \Sigma^* \to \Gamma^*$ so that:*

- $h(u \circ v) = h(u) \circ h(v)$ *for all $u, v \in \Sigma^*$, and*
- $h(\varepsilon) = \varepsilon$.

It suffices to define h on Σ. Then h is **uniquely defined** on Σ^*.

h operates on words in Σ^* by using symbol after symbol!

A morphism h is ε-free if $h(a) \neq \varepsilon$ for all $a \in \Sigma$.

Definition 1.4 ((Ext.) Regular Expressions $\mathfrak{Reg}_\Sigma$, $\mathfrak{Reg}_\Sigma^+$)

Regular expressions $\mathfrak{Reg}_\Sigma$ are defined as follows:

1. 0 is a **regular expression**.
2. For all $a \in \Sigma$ is a **a regular expression**.
3. If r and s are **regular expressions**, then so are
   - $(r + s)$ (union),
   - (rs) (concatenation),
   - (r^*) (Kleene star).

Extended regular expressions $\mathfrak{Reg}_\Sigma^+$ are obtained when we replace above "regular expression" by "extended regular expression" and also add

4. If r and s are **ext. regular expressions**, then so are
   - $(\neg r)$ (negation),
   - $(r \cap s)$.

We usually omit parentheses and agree that * is stronger than concatenation, which is stronger than $+$, $\cap$ and $\neg$.

Definition 1.5 (Semantics of $\mathfrak{Reg}_\Sigma^+$)

An (extended) regular expression r defines a language $\mathfrak{I}(r)$ over Σ as follows:

- $\mathfrak{I}(0) := \emptyset$,
- $\mathfrak{I}(a) := \{a\}$, for $a \in \Sigma$,
- $\mathfrak{I}(r + s) := \mathfrak{I}(r) \cup \mathfrak{I}(s)$,
- $\mathfrak{I}(r \cap s) := \mathfrak{I}(r) \cap \mathfrak{I}(s)$,
- $\mathfrak{I}(rs) := \mathfrak{I}(r)\mathfrak{I}(s)$,
- $\mathfrak{I}(r^*) := \mathfrak{I}(r)^*$,
- $\mathfrak{I}(\neg r) := \Sigma^* \setminus \mathfrak{I}(r)$.

We also use a^+ in a regular expression (this is a **macro**). We use "1", defined by $1 := 0^*$. Obviously $\mathfrak{I}(1) = \{\varepsilon\}$.
$\rightsquigarrow$ **blackboard 1.3**

1. What **added value** do $\neg$ and $\cap$ give us? Can we express more sets?
2. Consider the alphabet $\Sigma = \{a_0, a_1, \ldots, a_n\}$. Give a regular expression of the language consisting of all words over Σ that **have not the form** a_0^i for $i \geqslant 1$?
3. Give a regular expression of the language consisting of all words over Σ that **have not the following form**

$$(\ldots((a_0^2 a_1)^2 a_2)^2 \cdots a_n)^2.$$

Languages can be described more succinctly!

Given regular expression r, r', how **complex** is it to construct regular expressions for $\neg r$ and $r \cap r'$ and what is the **size** of these expressions?

1. **Naive approach**: Construct **dfa**'a (to compute the complement). This requires to transform a **nfa** in a dfa: exponential time (subset cnstructio. Indeed: **double exponential** (getting the regular expression from the automaton).
2. Also the size of the constructed regular expression is **double exponential**.

In fact, this is not accidental:

Size: In general, the size of the constructed regular expressions is **double exponential** in the input size.

Complexity: To construct $\neg r$ takes **double exponential time**. To construct $r \cap r'$ takes exponential time. (The case $r_1 \cap \ldots \cap r^k$ for unlimited k is also double exponential.)

We refer to Wouter Gelade and Frank Neven, *Succinctness of the Complement and Intersection of Regular Expressions*, STACS: 25th Annual Symposium on Theoretical Aspects of Computer Science, Bordeaux, pages 325–336, 2008.

Checking eex's (1)

Given a word w and an extended regular expression eex, how to efficiently determine that $w \in eex$?

1. Construct automata corresponding to **subexpressions. What about the complexity?**
2. Is there an algorithm polynomial in $|w| + |eex|$?

Checking eex's (2)

We construct a table with yes/no entries for

- each subexpression r of eex, and
- each substring x_{ij} of w (starting from position i and ending in j), such that

$$\begin{cases} Y, & \text{if } x_{ij} \in \mathfrak{I}(r); \\ N, & \text{else.} \end{cases}$$

$\rightsquigarrow$ **blackboard 1.4**

Checking eex's (3)

- The table has **size** $(|eex| + |w|)^3$ (n subexpr. of eex and $\frac{n(n+1)}{2}$ substrings of w).
- For each entry we have to consider all cases (to build the expression) and we can show, that determining each entry takes time at most $(|eex| + |w|)$. We start with small subexpressions $0, a$ of eex and all substrings of w. Then consider subexpressions built from $\cap, +, \circ, ^*, \neg$.
- This gives a **complexity of** $(|eex| + |w|)^4$.

Decidability of several problems

	Finite	Empty	Equivalence	Intersection
Type 0	no	no	no	no
Type 1	yes	no	no	no
0L	yes	no	no	no
Type 2	yes	yes	no	no
DCF	yes	yes	yes[a]	no
Type 3	yes	yes	yes	yes

[a] Very complicated, not in this lecture.

- yes/no entries: Already given in Informatics III.
- **yes/no**: This lecture.

Closure under certain operations

	$\circ$	$*$	$\neg$	h	h^{-1}	$\epsilon_{free}-h$	$\cup$	$\cap$	$\cap$ ***reg***	R
Type 0	✓	✓	–	✓	✓	✓	✓	✓	✓	✓
Rec	✓	✓	✓	–	–	✓	✓	✓	✓	✓
Type 1	✓	✓	✓	–	✓	✓	✓	✓	✓	✓
0L	–	–	–	–	–	–	–	–	–	–
Type 2	✓	✓	–	✓	✓	✓	✓	–	✓	✓
DCFL	–	–	✓	–	✓	–	–	–	✓	–
Type 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Deterministic finite automaton (dfa):

- Given $w \in \Sigma^*$, a dfa checks, whether $w \in L$.
- A dfa has:
  - a read-only head, moving to the left (not back).
  - an internal state, with finitely many values.
- It starts in the **initial state**.
- Reading one letter may result in a state change.
- If it has read the whole word and ends in a **final state**, then it accepts (“YES”): w is contained in L. Otherwise it does not accept: “No”.
- It stops after exactly $|w|$ steps.

Notation as a graph:

- a **node** for each possible **state**,
- **edges** are marked with symbols: They describe **state changes**.
- **Initial** states are marked with an **arrow**,
- **final states** are marked with a **double circle**.

Definition 1.6 (Deterministic Finite Automaton (dfa))

A deterministic finite automaton (dfa) is a tuple $A = (K, \Sigma, \delta, s_0, F)$, with

- K a finite set of **states**,
- Σ a **finite alphabet**,
- $\delta : K \times \Sigma \to K$ the (total) **transition function**,
- $s_0 \in K$ the **initial state**, and
- $F \subseteq K$ the set of **final states**.

$\delta(q,a) = q'$ means:

- The automaton is in state q,
- reads an a and
- reaches state q'.

We extend δ to $\delta^* : K \times \Sigma^* \to K$, which is defined recursively over Σ^* as follows:

$$\begin{aligned} \delta^*(q,\varepsilon) &:= q \\ \delta^*(q,wa) &:= \delta(\delta^*(q,w),a) \end{aligned}$$

We often write δ instead of δ^* if clear from context.

1.1 Makanin’s Algorithm

Content of this section:

1. We introduce finite sets of **word equations** over a set of constants Σ and a set of variables Ω. A solution is a **set of instantiations** of the variables.
2. **Makanin's algorithm** decides whether there are solutions for such a system and even computes them.

Example 1.7

Let $\Sigma = \{a, b, c\}$, $\Omega = \{x, y, z\}$ and consider the equation

$$abxcy = ycxba$$

Is that equation solvable? I.e. can we assign words from Σ^* to x, y such that we get the same words on both sides of the equation? How many solutions are there?

We denote the length of a word w by $|w|$, and the number of occurrences of a variable x in it by $|w|_x$. Thus $|abxcy| = 5$ and $|ycxba|_x = 1 = |ycxba|_y$.

Definition 1.8 (Word equations, solution)

A word equation over the alphabet Σ in the variables Ω is of the form

$$L = R,$$

where L, R are elements of $(\Sigma \cup \Omega)^*$.
A system of word equations over Σ in Ω is a set

$$S := \{L_1 = R_1, \ldots, L_k = R_k\}$$

of word equations. Such a system is quadratic if $\forall x \in \Omega : \sum_1^k (|L_i|_x + |R_i|_x) \leqslant 2$.

A solution of S is an instantiation $\sigma : \Omega \to \Sigma^*$ such that $\sigma(L_i) = \sigma(R_i)$ for all $i = 1, \ldots k$.

Theorem 1.9 (Makanin (1977))

There is an algorithm to compute whether there is a solution of a finite system of word equations over Σ in the variables Ω.

- Another formulation: the existential theory of equations over a free monoid is decidable.
- In 1983, the result was extended to the existential theory of equations over a free group is decidable.
- These are deep theorems and their proof is quite complicated. We discuss its relation to PCP and present a special case.

Lemma 1.10 (Quadratic Equations)

There is a simple algorithm due to Matijasevich (1968) to compute whether there is a solution of a finite system of quadratic *word equations over Σ in the variables Ω.*

Proof of Lemma 1.10 (1).

We assume the system E is given as k equations of the form $L_1 = R_1, \ldots, L_k = R_k$.

We show our lemma by **induction on Ω**.

For $\Omega = \emptyset$, the lemma is clear (simply checking whether all right and left hand sides correspond to each other). Let $\|E\| := \sum_{i=1}^{k} |L_i R_i|$

1. step: Set $\sigma(x) = \epsilon$ for some $x \in \Omega$, obtaining E' over $\Omega \setminus \{x\}$. If there is a solution (can be decided by induction hypothesis), we call it **singular solution**, we are done. If not, next step.

Proof of Lemma 1.10 (2).

2. step: Wlog we assume that for all solutions (if any) $\sigma(x) \neq \epsilon$ for all $x \in \Omega$. Then the first equation has either the form (1) $x \ldots = a \ldots$ or (2) $x \ldots = y \ldots$ where $x, y \in \Omega$, $x \neq y$, $a \in \Sigma$. In case (1) we transform $[x/az]$, in case (2) we transform $[x/yz]$, where z is a new variable.
We get a new system E', still quadratic, with: $\|E'\| \leqslant \|E\|$ and E **is solvable** **iff** E' **is solvable**.
Let σ be a non-singular solution of E where $\Sigma_{x\in\Omega}|\sigma(x)|$ is minimal. Then we find a solution σ' for E' with $|\sigma'(z)| \lneqq |\sigma'(x)|$ (note $\sigma(y) \neq \epsilon$). So the length of the shortest solution has decreased. We proceed recursively.

Proof of Lemma 1.10 (2).

3. step: It remains to *nondeterministically guess* a solution. If there is none, this can be detected because **there are only finitely many different systems** of equations possible (up to renaming of variables).

So we'll enter a loop eventually and know that we do not need to go on anymore.

Example 1.11 (Illustration of Lemma 1.10)

Let $\Sigma = \{a, b, c\}$ and $\Omega = \{x, y, z\}$ and consider

$$abxcy = ycxba$$

Figure 1 on slide 37 shows all the instances that we get (up to renaming of variables). Note that new equations need not have strictly less variables or strictly shorter length (**only the shortest solution length decreases**).

But they do not get any larger.

Therefore, there can be only finitely many of them. Thus if there is no solution at all, this can be determined. If there is a solution, it will be computed or guessed eventually.

We get as shortest length solution $x := a$, $y := aba$.
However, there is also the solution: $x := aca$, $y := aba$.

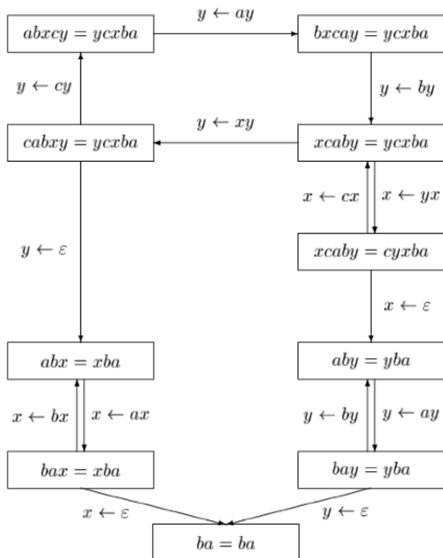

Figure 1: A simple algorithm for systems of **quadratic** equations.

What about the complexity of the general result?

Free monoids: The best bound of Makanin's algorithm is exponential space, while the prblem has been shown to be in **PSPACE**.

Free groups: There is no primitive recursive bound for Makanin's algorithm. However, the complexity of the problem has been shown to be in **PSPACE**.

1.2 Minimal DFA

Another method to determine whether a language is **not regular**. Given a language L, is there a unique minimal dfa for L?

Definition 1.12 ($\sim_L$)

For a language $L \subseteq \Sigma^*$ we define the relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$:

$$v \sim_L x \text{ if } \forall w \in \Sigma^* : (vw \in L \text{ \underline{\textbf{iff}} } xw \in L)).$$

Definition 1.13 (R_A)

For a dfa $A = (K, \Sigma, \delta, s_0, F)$ we define the relation $R_A \subseteq \Sigma^* \times \Sigma^*$:

$$vR_Ax \text{ if } \delta(s_0, v) = \delta(s_0, x).$$

Lemma 1.14

- $\sim_L$, R_A *are equivalence relations*
- $\sim_L$, R_A *are* *right congruences:*
  $(v \sim_L t$ ***iff*** $\forall w \in \Sigma^* : (vw \sim_L tw))$.

$\rightsquigarrow$ **blackboard 1.5**

Example 1.15

$L = \{w \in \{a,b\}^* : \#_a(w) = 1 \mod 3\}$.

There are exactly 3 equivalence classes:

1. $[\varepsilon] = \{w \in \{a,b\}^* : \#_a(w) = 0 \mod 3\}$.
2. $[a] = \{w \in \{a,b\}^* : \#_a(w) = 1 \mod 3\}$.
3. $[aa] = \{w \in \{a,b\}^* : \#_a(w) = 2 \mod 3\}$.

Example 1.16

$L = \{w \in \{a\}^* : \ w = a^p, \ p$ is prime$\}$.

There are **infinitely many equivalence classes**.

Example 1.17

$L = \{w \in \{a\}^* : \ w = a^p, \ p$ and $p + 2$ are primes$\}$.

How many equivalence classes are there?

Theorem 1.18 (Myhill-Nerode)

Let $L \subseteq \Sigma^*$ be a language. The following are equivalent:

(1) L is rational *(type 3).*

(2) L *is the* union *of some equivalence classes of a* right congruence of Σ^*, *which has only finitely many equivalence classes.*

(3) $\sim_L$ has only finitely many equivalence classes.

$\rightsquigarrow$ **blackboard 1.6**

Proof

(1) $\rightarrow$ (2): Let A be a dfa for L. Obviously R_A has only finitely many equivalence classes (bounded above by the number of states). Use Lemma 1.14 to show that it is a right congruence.

(2) $\rightarrow$ (3): We show: **Let R' be an equivalence relation which satisfies (2). Then each equivalence class of R' is contained in some equivalence class of $\sim_L$.**

Then $\sim_L$ has **at most** as many equivalence classes as R' (several classes of R' could be contained in $\sim_L$), therefore only finitely many.

Let $xR'y$. Because of the right congruence property (for all $z \in \Sigma^*$): $xzR'yz$. Thus $yz \in L$ **iff** $xz \in L$. Therefore $x \sim_L y$. Thus $[x]_{\sim_{R'}}$ is contained in $[x]_{\sim_L}$.

(3) → (1): $\sim_L$ is a right congruence (Lemma 1.14).

We construct a dfa $\hat{M}_L$, which accepts L:

- $K := \{[x]_{\sim_L} : x \in \Sigma^*\}$,
- $\delta([x]_{\sim_L}, a) = [xa]_{\sim_L}$,
- $s_0 := [\varepsilon]_{\sim_L}$,
- $F := \{[x]_{\sim_L} : x \in L\}$.

δ is well-defined (right congruence). Our dfa accepts L, because

$$\delta(s_0, x) = [x]_{\sim_L} \text{ thus } x \in L(\hat{M}_L) \text{ **iff** } [x]_{\sim_L} \in F$$

Definition 1.19 ($\hat{M}_L$)

For $L \subseteq \Sigma^*$ we define the following automaton $\hat{M}_L = (K, \Sigma, \delta, s_0, F)$:

- $K := \{[w]_{\sim_L} : w \in \Sigma^*\}$,
- $s_0 := [\varepsilon]_{\sim_L}$,
- $F := \{[w]_{\sim_L} : w \in L\}$,
- $\delta([w]_{\sim_L}, a) := [wa]_{\sim_L}$.

Is $\hat{M}_L$ a finite automaton?

Assume a dfa contains states q, q' with

$$\forall w \in \Sigma^* : \ \delta(q, w) \in F \ \underline{\textbf{iff}} \ \delta(q', w) \in F.$$

What does that mean?

q and q' are equivalent.

Does there exist to each dfa A another automaton which accepts the same language and has a minimal number of states?

Definition 1.20 (Morphisms between dfa's)

A **morphism between two dfa's** $A_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$, $A_2 = (K_2, \Sigma, \delta, s_2, F_2)$ is a mapping $h : K_1 \to K_2$ such that:

1. $h(s_1) = s_2$,
2. $h(F_1) \subseteq F_2$,
3. $h(K_1 \backslash F_1) \subseteq K_2 \backslash F_2$,
4. $h(\delta_1(q, a)) = \delta_2(h(q), a)$ for all $q \in K_1$ and $a \in \Sigma$.

An **isomorphism between two dfa's** is a bijective morphism between them.

In the following we assume that all states of an automaton are reachable.

Lemma 1.21

For any morphism h from dfa A_1 to dfa A_2 it holds that $L(A_1) = L(A_2)$.

Proof

Let s_i be the initial and F_i the set of final states of A_i.

$$
\begin{aligned}
w \in L(A_1) &\Leftrightarrow \delta_1(s_1, w) \in F_1 \\
&\Leftrightarrow h(\delta_1(s_1, w)) \in F_2 \\
&\Leftrightarrow \delta_2(h(s_1), w) \in F_2 \\
&\Leftrightarrow \delta_2(s_2, w) \in F_2 \\
&\Leftrightarrow w \in L(A_2)
\end{aligned}
$$

Lemma 1.22

The mapping $h_A : A \to \hat{M}_{L(A)}$ *defined by* $h_A(q) = [w]_{\sim_{L(A)}}$ *if* $\delta(s_0, w) = q$ *is a* **surjective morphism** *from* A *to* $\hat{M}_{L(A)}$.

Proof.

- Firstly, we observe that h_A is a **function** as A is reachable.
- h_A is **well-defined**: Suppose there is another word $v \neq w$ with $\delta(s_0, v) = q$. Then, $v \sim_{L(A)} w$ and thus $[v] = [w]$.

- h_A is a **morphism**: $h_A(s_0) = [\epsilon]$ is the initial state of $\hat{M}_{L(A)}$.
  Final states: Let $q \in F_A$ **iff**
  $\exists w \in L(A)(\delta(s_0, w) = q)$ **iff**
  $\exists w \in L(A)(h_A(q) = [w])$ **iff** $h_A(q) \in F_{\hat{M}_{L(A)}}$.
  Transitions: $\hat{\delta}(h_A(q), a) = \hat{\delta}([w], a)$ with $\delta(s_0, w) = q$ which is equivalent to $[wa]$ which is equivalent to $h_A(p)$ with $p = \delta(s_0, wa) = \delta(\delta(s_0, w), a) = \delta(q, a)$, which is thus equal to $h_A(\delta(q, a))$.
- h_A is **surjective**: For each w there is a state q with $\delta(s_0, w) = q$.

Theorem 1.23 (Minimality of $\hat{M}_L$)

Let A be a dfa and $L = L(A)$. The following are equivalent:

1. *A is* **minimal***: each dfa A' with $L(A') = L(A)$ has at least as many states as A.*
2. **A is isomorphic to $\hat{M}_L$.**

Proof of Theorem 1.23

We have shown (Proof of Theorem 1.18), that each other dfa accepting L, defines an equivalence relation which is a **refinement of** $\sim_L$: each of these classes is contained in one of the classes of $\sim_L$.
Therefore each such dfa has at least as many classes as $\hat{M}_L$.

Let M' be a dfa which accepts L and has the same number of states as $\hat{M}_L$. **We define an isomorphism between M' and $\hat{M}_L$.** Let q' a state of M'. Then there is a word w with $\delta'(s'_0, w) = q'$ (otherwise q' could be removed). We map q' to $\delta(s_0, w)$ (this is well-defined). □

Proof of Theorem 1.23 (cont.)

"$1 \Rightarrow 2$": Let A be minimal. There is a surjective moprhism $h_A : A \to \hat{M}_L$ (Lemma 1.22). Now, if h_A would not be injective we have that A has more states than $\hat{M}_L$. Contradiction as A is minimal!
"$2 \Rightarrow 1$": Let A be isomorph to $\hat{M}_L$; hence, there is a bijection $h'_A : \hat{M}_L :\to A$. Let A' be a (reachable) dfa with $L(A') = L$. By Lemma 1.22 there is a surjective morphism $h_{A'} : A' \to \hat{M}_L$. Now we have that $h_{A'A} := h'_A \circ h_{A'} : A' \to A$ is a surjective morphism. Then, we have that $|K_{A'}| \geqslant |K_A|$ which implies that A is minimal as A' is arbitrary.

Definition 1.24 (Equivalence of states)

Given a dfa A, we call two states q_1, q_2 **equivalent**, if $\forall w \in \Sigma^* : \ \delta(q_1, w) \in F$ **iff** $\delta(q_2, w) \in F$

How can we check equivalence of states? After all, we have to take into account **infinitely many words** w.

Check words of length **at most** $|K| - 2$!

Theorem 1.25 (Finite test set)

If two states are equivalent for all words of length at most $|K| - 2$, then they are equivalent in the sense of Definition 1.24.

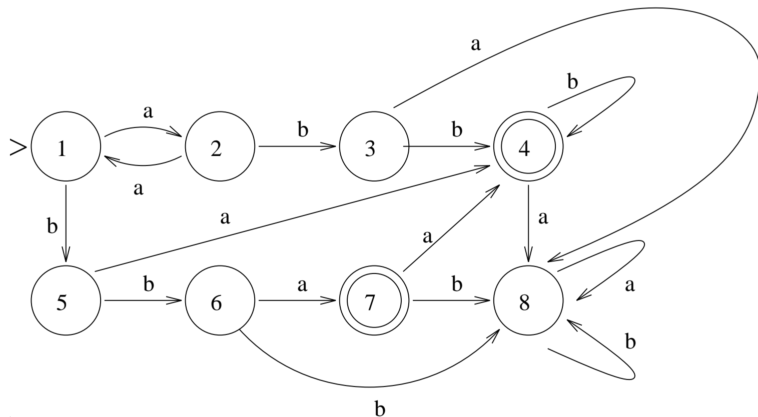

Figure 2: A dfa A.

We first check words of length at most 0, then 1, then 2, $\ldots$, until 6.

Definition of $\sim_n$: $\rightsquigarrow$ **blackboard 1.7**

Lemma 1.26

Let A be a reachable dfa. Let E_n denote the partition belonging to $\sim_n$ and E belonging to $\sim$.

1. E_{n+1} *and* E *are* *finer* *than* E_n.
2. $q \sim_{n+1} q'$ ***iff*** *(1)* $q \sim_1 q'$ *and (2) for all* $a \in \Sigma : (\delta(q,a) \sim_n \delta(q',a))$ *for all* $q, q' \in K$ *and* $n \in \mathbb{N}$.
3. *If* $E_n = E_{n+1}$ *then* $E_n = E$.
4. $E_{n_0} = E$ *for* $n_0 = |K| - 2$.

This lemma implies Theorem 1.25.

Lemma 1.27

Let A be a dfa accepting the language L with the following two properties:

1. *All states are reachable.*
2. *No two states are equivalent.*

Then A is isomorphic to $\hat{M}_L$.

Proof.

Let A be a reachable dfa in which no two states are equivalent. By Lemma 1.22 there is a surjective morphism $h_A : A \to \hat{M}_L$. It remains to show that h_A is also bijective. Suppose h_A is not injective. Then there are states q, q' with $h_A(q) = h_A(q')$: □

1.3 cf = hom(Dyck∩reg)

We introduced in Informatik III the Dyck languages Dyck_n.

Chomsky-Schützenberger

Our main theorem in this section says, that a contextfree language is essentially the homomorphic image of a Dyck language.

Later (Theorem 1.73 on Slide 163) we prove that a type 0 language is essentially the homomorphic image of a type 1 language.

Definition 1.28 (Dycklanguage D_k)

For $k \in \mathbb{N}$ let $V_k := \{x_1, \overline{x_1}, x_2 \ldots, x_k, \overline{x_k}\}$ an alphabet over the $2k$ symbols $x_1, \overline{x_1}, \ldots, x_k, \overline{x_k}$.
The **Dyck language** D_k is the smallest set satisfying the following conditions:

1. $\epsilon \in D_k$,
2. If $w \in D_k$, then so is $x_i w \overline{x_i}$.
3. If $u, v \in D_k$, then so is uv.

This set can be seen as the set of all correctly formed expressions with brackets.

Theorem 1.29 (Closed under hom)

The classes $\mathcal{L}_0$, $\mathcal{L}_2$, $\mathcal{L}_3$ *are* closed under homomorphisms. $\mathcal{L}_1$ *is* closed under ε-free homomorphisms. *All classes are* closed under inverse homomorphisms.

Proof

Let $G_i = (V_i, T_i, R_i, S_i)$ with $i \in \{1, 2\}$ two cf grammars, $L_i = L(G_i)$ and $V_1 \cap V_2 = \emptyset$.
Let $h : T_1^* \to \Gamma^*$ a morphism, and let $G := (V_1, \Gamma, R_1', S_1)$; with $A \to \alpha \in R_1$ **iff** $A \to \hat{\alpha} \in {R_1}'$, and $\hat{\alpha}$ is obtained from α by replacing each occurrence of a terminal a in α by $h(a)$. Then G generates exactly $h(L_1)$.
The remaining parts follow easily. □

Theorem 1.30 (CFL= h(Dyck ∩ reg))

For each **contextfree language** L *there is a* **n** $\in \mathbb{N}$, *a regular language* L_{reg} *and a homomorphism* h *with*

$$L = h(\mathrm{Dyck}_n \cap L_{reg})$$

Proof

Let L be given by $G = (V, T, R, S)$ in Chomsky normalform $L = L(G)$. We set $n := |R|$.
We assume wlog that the first l rules have the form $A \Longrightarrow BC$ (denoted by $r_i : A_i \Longrightarrow B_i C_i$, $1 \leqslant i \leqslant l$) and the remaining rules have the form $A \Longrightarrow a$ (denoted by $r_j : D_j \Longrightarrow a_j$, $l+1 \leqslant j \leqslant n$). Note that these symbols are not all distinct: all A_i, B_i, C_i's are also some D_j's (**why**?).
And not all a_j's have to be different:
$V = \{A_i, D_j | 1 \leqslant i \leqslant l \lneqq j \leqslant n\}$ and $T = \{a_j | l \lneqq j \leqslant n\}$.
We define the following $l + (l+1)(n-l)$ rules:

1. $A_i \Longrightarrow x_i B_i$, for $1 \leqslant i \leqslant l$,
2. $D_j \Longrightarrow x_j \overline{x_j x_i} C_i$, for $1 \leqslant i \leqslant l$ and $l+1 \leqslant j \leqslant n$,
3. $D_j \Longrightarrow x_j \overline{x_j}$, for $l+1 \leqslant j \leqslant n$.

So we get a new grammar $G' = (V, V_n, R', S)$.

Proof (2)

The terminals a_j from G are now the words $x_j\overline{x_j}$. What is the difference between $L(G)$ and $L(G')$?
As it is rightlinear, the language generated is **regular**.
Define L_{reg}:=$L(G')$. There are words generated that are not in Dyck_n.
Not all words generated by G' correspond to derivations in G. The problem is that the parentheses x_i and $\overline{x_i}$ might not match (and one cannot enforce this by the pumping lemma). However, the matching expressions are all contained in $L(G')$. Indeed, we claim that this is the only difference:

Proof (3)

We show:

1. For each derivation in G of a word w, there is a corresponding derivation in G' which is also in Dyck_n.
2. Each derivation in G' of a word w' which is also in Dyck_n, comes from a corresponding derivation in G of the corresponding word w'.

In an exercise, you have to make precise the meaning of corresponding. This immediately leads to a homomorphism h such that $L = h(\mathrm{Dyck}_n \cap L_{reg})$.

Up to now, we did not use the automaton that corresponds to a cf grammar.

1. cf languages are covered by PDA'a (push-down automata) (see Informatik III).
2. We also repeat the following undecidability result from Informatik III. The problem to decide for a given DTM M whether $L(M) = \emptyset$ is undecidable.
3. Remember Ogden's lemma.
4. Remember the Chomsky normalform.

From Informatics III

Classical definition of a PDA.

Idea of the **Stack, batch**: Last in, first out:
it can store arbitrarily many information units; only the most recent info can be accessed.

- A rule $S \rightarrow aAb$ is like a **call** of a procedure for A: after processing A there is still a b to process.
- **Push-Down-Automaton**: like finite automaton, but with an additional stack. Transition function:
  - depends on the first stack symbol
  - does not only change the state, but also reads and writes on the stack

Definition 1.31 (Push-Down-Automaton (PDA))

A Push-Down-Automaton (PDA) is a tuple $M = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ with

K	a finite set of states
Σ	the input alphabet
Γ	the stack alphabet
$s_0 \in K$	the initial state
$Z_0 \in \Gamma$	the downmost stack symbol
$F \subseteq K$	a set of final states
Δ	the transition function, a finite subset of $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (K \times \Gamma^*)$

What does the transition function?

In one working step, the PDA does

- depending on the actual state K,
- depending on the next input symbol (or not depending on it),
- depending on the topmost stack symbol

the following:

- next input symbol is read, or not (reading ε),
- topmost stack symbol is removed,
- state might be changed
- zero or more symbols are written on the stack. For a new stack word $\gamma = A_1 \dots A_n$, A_1 will be topmost, then A_2 etc until A_n.

Used symbols:

- Symbols a, b, c for letters from Σ,
- u, v, w for words from Σ^*,
- A, B for stack symbols of Γ,
- γ, η for whole stacks from Γ^*.

What does the transition function?

- Notion of configuration: describes the actual situation of the PDA **completely**
- Consists of:
  - **current state**,
  - **rest of the input**,
  - **complete stack**
- Relation $\vdash$ between configurations:
  For configurations C_1, C_2 the string

$$C_1 \vdash C_2$$

  means that the **PDA can get from C_1 to C_2 in one single step**.

Definition 1.32 (Accepted language of a PDA)

A PDA M can accept a language in two different ways: using final states or using empty stack:

$$L_f(\mathrm{M}) = \{w \in \Sigma^* \mid \exists q \in F\ \exists \gamma \in \Gamma^* \ \big((s_0, w, Z_0) \vdash^*_{\mathrm{M}} (q, \varepsilon, \gamma)\big)\}$$

$$L_e(\mathrm{M}) = \{w \in \Sigma^* \mid \exists q \in K \ \big((s_0, w, Z_0) \vdash^*_{\mathrm{M}} (q, \varepsilon, \varepsilon)\big)\}$$

Notes:

- If clear from context whether we mean $L_f(\mathrm{M})$ or $L_e(\mathrm{M})$, we simply write $L(\mathrm{M})$.
- The input w must be scanned M completely:

$$(s_0, w, Z_0) \vdash^* (q, \varepsilon, \cdot)$$

  is requested.
- A PDA might not terminate (even if it is accepting).

Note that the bottommost symbol may be removed, but then *the PDA hangs*: it can not work anymore, *there is no successor configuration*.

This corresponds to the *hanging of a nd. fa*, if the transition function is not defined (which can happen here as well).

Definition 1.33 (Configuration of a PDA, $\vdash$)

A configuration C of a PDA $M = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ is a triple

$$C = (q, w, \gamma) \in K \times \Sigma^* \times \Gamma^*.$$

(w is the part of the input that has not yet been read, γ is the complete stack, and q is the actual state.)

(s_0, w, Z_0) is called initial configuration with input w.

C_2 is called successor configuration of C_1, or $C_1 \vdash C_2$, if $\exists a \in \Sigma\ \exists A \in \Gamma\ \exists w \in \Sigma^*\ \exists \gamma, \eta \in \Gamma^*$, so that

either $C_1 = (q_1, aw, A\gamma)$, $C_2 = (q_2, w, \eta\gamma)$, and $(q_1, a, A)\ \Delta\ (q_2, \eta)$,

or $C_1 = (q_1, w, A\gamma)$, $C_2 = (q_2, w, \eta\gamma)$, and $(q_1, \varepsilon, A)\ \Delta\ (q_2, \eta)$,

- Δ of a PDA is essentially indeterministic:

$$\Delta \subseteq (K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (K \times \Gamma^*)$$

Using the indeterminism is a powerful tool:

(Palindroms over $\{a, b\}$)

$(s_0, \varepsilon, Z_0) \ \Delta \ (s_1, \varepsilon)$ } accepting ε

$$\left.\begin{array}{ll}
(s_0, a, Z_0) & \Delta\ (s_0, A) \\
(s_0, a, A) & \Delta\ (s_0, AA) \\
(s_0, a, B) & \Delta\ (s_0, AB) \\
(s_0, b, Z_0) & \Delta\ (s_0, B) \\
(s_0, b, A) & \Delta\ (s_0, BA) \\
(s_0, b, B) & \Delta\ (s_0, BB)
\end{array}\right\} \text{Build the stack}$$

$$\left.\begin{array}{ll}(s_0,\varepsilon,A) & \Delta\,(s_1,\varepsilon)\\ (s_0,\varepsilon,B) & \Delta\,(s_1,\varepsilon)\end{array}\right\}\text{ Change direction for palindromes with odd number of symbols}$$

$$\left.\begin{array}{ll}(s_0,a,A) & \Delta\,(s_1,\varepsilon)\\ (s_0,b,B) & \Delta\,(s_1,\varepsilon)\end{array}\right\}\text{ Change direction for palindromes with even number of symbols}$$

$$\left.\begin{array}{ll}(s_1,a,A) & \Delta\,(s_1,\varepsilon)\\ (s_1,b,B) & \Delta\,(s_1,\varepsilon)\end{array}\right\}\text{ Remove the stack}$$

Definition 1.34 (Turing-Machine (DTM))

A **deterministic Turing-Machine (DTM)** M is a tuple $M = (K, \Sigma, \delta, s)$ with

- K a finite set of states, $h \notin K$ (h is the **final state**),
- Σ an alphabet with $L, R \notin \Sigma, \# \in \Sigma$,
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ a transition function, and
- $s \in K$ a start state.

Number of states: $|K| - 1$ (s is not counted).

Definition 1.35 (Configuration of a DTM, ID (1st try))

A *configuration* or *instantaneous description* *(ID)* C of a DTM $M = (K, \Sigma, \delta, s)$ is a word of the form $C = wqu$ with

- $q \in K \cup \{h\}$ is the actual state,
- $w \in \Sigma^*$ the contents of the tape left of the head,
- $u \in \Sigma^*$ the contents of the tape right of the head, the head is located on the leftmost symbol of u (if $u = \epsilon$ then the blank is scanned).

Definition 1.36 (Successor Configuration)

A configuration C_2 is called **successor configuration** of C_1, written

$$C_1 \vdash_M C_2$$

if:

- $C_i = w_i q_i a_i u_i$ for $i \in \{1, 2\}$, and
- there is a transition $\delta(q_1, a_1) = \langle q_2, b\rangle$ as follows:

  Case 1: $b \in \Sigma$. Then $w_1 = w_2,\ u_1 = u_2,\ a_2 = b$.

  Case 2: $b = L$. Then for w_2 and a_2: $w_1 = w_2 a_2$.
  For u_2: If $a_1 = \#$ and $u_1 = \epsilon$ then ist $u_2 = \epsilon$, otherwise $u_2 = a_1 u_1$.

  Case 3: $b = R$. Then $w_2 = w_1 a_1$.
  For a_2 and u_2: If $u_1 = \epsilon$ then $u_2 = \epsilon$ and $a_2 = \#$, otherwise $u_1 = a_2 u_2$.

Definition 1.37 (Valid Computation)

Let M be a DTM. We write

$$C \vdash_{\mathrm{M}}^{*} C'$$

iff there is a sequence $C_0, C_1, \ldots, C_n$ of configurations ($n \geqslant 0$), s.t. $C = C_0$ and $C' = C_n$ and for all $i < n$: $C_i \vdash_{\mathrm{M}} C_{i+1}$.
Then $C_0, C_1, \ldots, C_n$ is called a valid computation of M of length n from C_0 to C_n.
A computation halts (or terminates), if the last configuration does.

In view of some undecidability results in this section, we need some results about (in-)valid DTM computations.

We have just described the concept of a valid computation. However, for our purposes we redefine it slightly: a valid computation is a string of the form

$$w_1 \# w_2^R \# w_3 \# w_4^R \# \dots \# w_n$$

such that the usual conditions hold (each w_i is a configuration, w_1 the initial configuration, and w_{i+1} is a successor configuration of w_i).

The reason why we are using w_2^R instead of w_2 is because $\{ww^R | \, w \in \Sigma^*\}$ is contextfree, but $\{ww | \, w \in \Sigma^*\}$ is not.

Definition 1.38 (Arbitrary computations $Comp(M)$)

For a TM M, the set of arbitrary computations, denoted by $Comp(M)$, consists of all (finite) strings of the form $w_1\#w_2^R\#w_3\#w_4^R\#\dots\#w_n$ such that:

1. w_1 is an initial ID of M (i.e. M starts on some word w),
2. all w_i are ID's compatible to M,
3. $w_i \vdash w_{i+1}$: the ID's describe a valid step of M.

We do not yet assume that w_n is a terminating ID: the computation might go on forever.
$Comp_{\text{halt}}(M)$ is the subset of $Comp(M)$ which also satisfies:

4. w_n is the final ID: M halts with w_n.

1. Note that $Comp_{\text{halt}}(M)$ consists of terminating computations.
2. The general halting problem for TM's is closely related to $Comp_{\text{halt}}(M)$: M does not terminate for any word w ($L(M) = \emptyset$) **iff** $Comp_{\text{halt}}(M) = \emptyset$.
3. M terminates on all words w ($L(M) = \Sigma^*$) **iff** $Comp_{\text{halt}}(M) = \Sigma^*$.
4. The last two properties are not true for $Comp(M)$.

Theorem 1.39 ($Comp_{\text{halt}}(\mathrm{M}) = L_{cf} \cap L'_{cf}$)

The set $Comp_{halt}(\mathrm{M})$ of a DTM M is the intersection of two cf languages *(and grammars for them can be constructed).*
The same is true for $Comp(\mathrm{M})$.

Proof.

The idea is to use L_1 for $w_i \vdash w_{i+1}^R$ for odd i and L_2 for $w_i^R \vdash w_{i+1}$ for even i.
The intersection of both languages is then the set of all computations.
Complete Proof: $\rightsquigarrow$ **blackboard 1.7**. □

Theorem 1.40 (Invalid TM computations as a CFL)

The set of all invalid halting computations of a DTM*, i.e. the complement of* $Comp_{\mathit{halt}}(\mathrm{M})$*, is a* cf language *(and its grammar can be constructed). The same is true for the complement of* $Comp(\mathrm{M})$.

Proof.

When is a string **not** in $Comp_{\text{halt}}(M)$?

1. It is not of the required form $w_1\#w_2\#\ldots$, or
2. w_1 is not initial, or
3. w_n is not final, or
4. $w_i \vdash w_{i+1}^R$ is false for some odd i, or
5. $w_i^R \vdash w_{i+1}$ is false for some even i.

The first 3 properties can be decided by dfa's. The last 2 properties can be decided by PDA's.

The set of invalid computations is then the union of two cf languages and a regular set, thus it is itself cf. □

We use Theorems 1.39 and 1.40 for several undecidability results.

Theorem 1.41 (Undecidability for CFL Problems)

The following problems are undecidable. *For cf grammars G_1, G_2 and R a regular set, to decide*

1. *Is* $L(G_1) \cap L(G_2) = \emptyset$*?*
2. *Is* $L(G_1) = \Sigma^*$*?*
3. *Is* $L(G_1) = L(G_2)$*?*
4. *Is* $L(G_1) \subseteq L(G_2)$*?*
5. *Is* $L(G_1) = R$*?*
6. *Is* $R \subseteq L(G_1)$*?*

Proof.

1. Reduce $L(M) = \emptyset$ to the problem.
2. For any TM M, we construct (using Theorem 1.40) a cf grammar G with $L(G) = \Sigma^*$ **iff** $L(M) = \emptyset$.
3. Let G_2 by any grammar that generates Σ^*. Reduce to 1.
4. Let G_2 by any grammar that generates Σ^*. Reduce to 1.
5. Let $R = \Sigma^*$ and reduce to 1.
6. Let $R = \Sigma^*$ and reduce to 1.

Here is another theorem that helps us to establish some undecidability results.

Theorem 1.42 (Valid Computations of a TM)

Let M be a DTM that makes at least 2 moves *on every input. Then the following are equivalent:*

- *The set $Comp_{\mathit{halt}}(\mathrm{M})$ is cf.*
- *$L(\mathrm{M})$ is finite.*

Proof: (2) implies (1): trivial. For the other one, assume the set is cf and infinite. Then one can choose a word $w_1\#w_2^R\#w_3\#\ldots$ s.t. w_2 is greater than the constant from Ogden's lemma (for the cf language $Comp_{\mathrm{halt}}(\mathrm{M})$). Find an appropriate marking and pump once to pump the word out of the language.
$\rightsquigarrow$ **blackboard 1.8**.

From Informatik III: Ogden’s Lemma

Theorem 1.43 (Ogden’s Lemma)

For each **cf language** $L \subseteq \Sigma^*$ *there exists* $n \in \mathbb{N}$, *such that: For all* $z \in L$ *with* $|z| \geqslant n$ *and each* **marking** *of at least* n *distinguished positions in* z *(connected or not) there exist* $u, \boldsymbol{v}, w, \boldsymbol{x}, y, \in \Sigma^*$, *s.t.*

- $z = u\boldsymbol{v}\, w\, \boldsymbol{x}y$,
- $\boldsymbol{v}$ *and* $\boldsymbol{x}$ *contain all together at least one distinguished position,*
- $\boldsymbol{v}w\,\boldsymbol{x}$ *contains at most* n *distinguished positions,*
- $u\boldsymbol{v}^i w\boldsymbol{x}^i y \in L$ *for all* $i \in \mathbb{N}_0$.

We use Theorem 1.42 to prove:

Theorem 1.44 (Undecidability for CFL Problems (2))

The following problems are undecidable. *For cf grammars* G_1, G_2*:*

1. *Is* $\overline{L(G_1)}$ *cf?*
2. *Is* $L(G_1) \cap L(G_2)$ *cf?*

Proof: ⇝ **blackboard 1.9**.

What about the problem whether $L(G_1)$ is finite?

Here are a few problems for cf languages that are decidable.

Lemma 1.45 (Decidable cf problems)

The following problems are decidable for a contextfree grammar G:

1. *Is $L(G) = \emptyset$?*
2. *Is $L(G)$ finite?*
3. *Is $L(G)$ infinite?*

Proof: Think of useless symbols, and Chomsky Normalform and the corresponding problems for regular languages.

1.4 DPDA/DCF

Deterministic PDA's with a deterministic, but not necessarily complete relation.

- For each combination of state, input- and stack symbol, there is at most one transition.
- If the PDA can do a ε-transition, then it is not allowed **to read alternatively an input symbol**.

Definition 1.46 (DPDA)

A PDA $M = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ is **deterministic**, if:

- $\exists a \in \Sigma \;\; \Delta(q, a, Z) \neq \emptyset \implies \Delta(q, \varepsilon, Z) = \emptyset$ for all $q \in K, Z \in \Gamma$: If in a state q with top stack symbol Z a symbol a can be read on the input tape, then **there is no ε-transition** for q and Z.
- $|\Delta(q, a, Z)| \leqslant 1$ for all $q \in K, Z \in \Gamma, a \in \Sigma \cup \{\varepsilon\}$: There is **at most one transition** in each configuration.

The class of languages that can be accepted by DPDA's is denoted by **DCF**. We also call such a language simply **dcf**.

Note that $\Delta(q, a, Z)$ can be empty, **even if the input is not yet consumed completely**. In that case, ε-moves are possible, until a state or topmost stack symbol is reached such that $\Delta(q', a, Z') \neq \emptyset$.

Theorem 1.47 ($\mathcal{L}_3 \subsetneqq$ DCF $\subsetneqq \mathcal{L}_2$)

The following holds:

$$\mathcal{L}_3 \subsetneqq \textbf{\textit{DCF}} \subsetneqq \mathcal{L}_2$$

when acceptance *for DPDA's is defined with* final states.

For DPDA's the **acceptance with empty stack** *is* **weaker** *than with final states (in fact, it does not make sense, as even $\mathcal{L}_3 \subseteq$* ***DCF*** *does not hold).*

Proof

DCF $\subsetneqq \mathcal{L}_2$: "$\subseteq$" is trivial. We show that $L := \{a^i b^j c^k \mid i = j \text{ or } j = k\} \in \mathcal{L}_2 \setminus$ **DCF**. As L is contextfree (guess nondeterministically whether $i = j$ or $j = k$) and **DCF** is closed under complements (see below), then so would be $\overline{L} = (abc)^* \setminus L$ (if $L \in$ **DCF**). But then $\overline{L} \cap a^*b^*c^*$ would also be contextfree (intersection of a cf with a regular language). But we have:

$$\overline{L} \cap a^*b^*c^* = \{a^i b^j c^k \mid i \neq j \text{ and } j \neq k\}$$

and this language is not contextfree (Pumping Lemma).

$\mathcal{L}_3 \subsetneq$ **DCF:** "$\subseteq$" is trivial, because each det. PDA can be used as a dfa.
The language $\{wcw^R \mid w \in \{a,b\}^*\}$ is not regular, but in **DCF** (no need to guess the middle, as this is given by c).

Empty stack vs final states: Consider $L := \{a, ab\}$ over $\Sigma = \{a, b\}$. As a has to be accepted via empty stack, there is a computation for each DPDA of the form

$$(q_0, a, Z_0) \ \vdash^* \ (q, \epsilon, \epsilon)$$

Each such DPDA does the following on ab:

$$(q_0, ab, Z_0) \ \vdash^* \ (q, b, \epsilon)$$

thus it **hangs**.
Therefore no DPDA accepting via empty stack can accept the language $L := \{a, ab\}$.
It is easy to construct DPDA's accepting this language via final states.

- It seems that the difference between accepting with final states and with empty stack depends on the fact that our PDA *gets stuck* when the stack is empty (and the word to be scanned is not yet fully read).
- So why not simply allow the transition function to be defined on $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \cup \{\epsilon\})$ rather than on $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma)$.
- Then a PDA does not get stuck when the stack is empty (the successor configuration can be changed easily) and our counterexample does not apply.

Can you find a different counterexample?

DCF is closed under complements, because it is deterministic. But this is nontrivial, because a DPDA might

- **hang,** and then the complement automaton (where final and nonfinal states have been switched) would hang as well: **a word would then be accepted in neither automaton**;
- **have ϵ-moves,** which lead from a final in a nonfinal state. Then **a word w would be accepted from both automata**.

DPDA's are allowed to do ϵ-moves (but not, when at the same time a symbol can be scanned). **What if we do not allow ϵ-moves at all?**

Lemma 1.48 (DPDA without ϵ-moves)

1. *DPDA's without ϵ-moves accept less languages than DPDA's with ϵ-moves.*
2. *However, one can restrict wlog. to DPDA's* without ϵ-moves after the last symbol is read.

Proof

For 1., consider the language

$$\{0^i 1^j a 2^i \,|\, i, j \geqslant 1\} \cup \{0^i 1^j b 2^j \,|\, i, j \geqslant 1\}$$

For 2., use the notion of a predicting machine.

The predicting machine

For each DPDA M and dfa A we can construct a DPDA $\pi(M, A)$ such that:

1. $(q_0, x, [Z_0, \mu_0]) \vdash^*_{\pi(M,A)} (r, y, [Z_1, \mu_1][Z_2, \mu_2] \ldots [Z_n, \mu_n])$
   iff
   1. $(q_0, x, Z_0) \vdash^*_M (r, y, Z_1 \ldots Z_n)$ and
   2. for $1 \leqslant i \leqslant n$: μ_i is the set of all (q, p) for which

      for some w $(q, w, Z_i Z_{i+1} \ldots Z_n) \vdash^*_M (s, \varepsilon, \gamma)$

      for some $s \in F_M$ and $\gamma \in \Gamma^*$ and $\delta_A(p, w) \in F_A$.

So the new stack symbols give us information about DPDA M and dfa A: is there some input string that causes both M and A to terminate on it.

Theorem 1.49 (DPDA do not hang)

For each DPDA M there is an equivalent DPDA M′ which does not hang: it scans the entire input (for all inputs).

Lemma 1.50 (DCF is closed under complementation)

*The complement of a **DCF** language is also a **DCF** language.*

Lemma 1.51 (DCF $\cap$ Regular = DCF)

*The intersection of a **DCF** language with a regular language is in **DCF**.*

Proof: Exactly the same construction as for a PDA.

Lemma 1.52 (DCF not closed under operations)

***DCF** is not closed under $\cup$, $\cap$, homomorphisms, concatenation, Kleene closure.*

(Part 1).

$\cup, \cap$: Consider the proof in Informatik III, that $\mathcal{L}_2$ is not closed under $\cap$: the languages used there are all in **DCF**. As **DCF** is closed under complements, it can not be closed under $\cup$ because we have

$$L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$$

concatenation: Consider $L_1 = \{a^i b^i c^j : i, j \geqslant 0\}$, $L_2 = \{a^i b^j c^j : i, j \geqslant 0\}$, and $L_3 = 0L_1 \cup L_2$, which is **DCF**, but $0^* L_3$ is not. **Assume it is.** Then $L_4 = 0^* L_3 \cap 0a^*b^*c^*$ is also **DCF**. But $L_4 = 0L_1 \cup 0L_2$, so $L_1 \cup L_2$ is also **DCF**: Contradiction.

(Part 2).

homomorphism: $L_5 := 0L_1 \cup 1L_2$ is **DCF**. Let h be the hom that maps 1 to 0 and leaves all other symbols as they are. Then $h(L_6) = L_4$, which is not **DCF**.

Kleene closure: $L_6 := \{0\} \cup L_3$ is **DCF**, but L_6^* is not (why?).

Theorem 1.53 (Decidability of some DCF problems)

Let L be dcf and R regular. Then the following problems are decidable:

1. *Is $L = R$?*
2. *Is $R \subseteq L$?*
3. *Is $L \subseteq R$?*

Compare with Theorem 1.41.

Proof.

1. $L = R$ **iff** $L_1 = (L \cap \overline{R}) \cup (\overline{L} \cap R) = \emptyset$. But L_1 is cf. Eliminate useless symbols. If useful symbols remain, the language is not empty.
2. $R \subseteq L$ **iff** $\overline{L} \cap R = \emptyset$. Again, $\overline{L} \cap R$ is cf.
3. $L \subseteq R$ **iff** $\overline{R} \cap L = \emptyset$. Again, $\overline{R} \cap L$ is cf.

Theorem 1.54 (Undecidability of some DCF problems)

The following problems are **undecidable**. *For dcf* L, L':

1. *Is* $L \cap L' = \emptyset$?
2. *Is* $L \subseteq L'$?
3. *Is* $L \cap L'$ *dcf?*
4. *Is* $L \cap L'$ *cf?*
5. *Is* $L \cup L'$ *dcf?*

Proof.

1. Observe that the languages used in Theorem 1.39 are in fact **DCF**.
2. From 1. by $L \subseteq L'$ **iff** $L \cap L' = \emptyset$.
3. Use Theorem 1.42. Then the intersection is either finite or not cf. Thus decidability of the problem in question would imply decidability of finiteness for $L(M)$.
4. Same as for 3.
5. Closure under complementation: reduces 3. to 5.

- We will show later (Theorem 2.14) that it is undecidable to determine, given any contextfree grammar G, whether the language generated by G is deterministic contextfree. Compare with Theorem 1.41, 5.
- However, the problem to determine whether two dcf languages, represented by their contextfree grammars, are identical, is decidable.

Theorem 1.55 (Senizergues, 1997)

Given two DPDA's, it is decidable whether they generate the same language. The upper bound on the decision procedure is primitive recursive.

Theorem 1.56 (Valiant, 1966)

Given a dcf language L, it is decidable whether L is regular or not.

We'll show later (Corollary 2.14) that if we only assume that L is cf, then it is undecidable whether L is regular.

1.5 LBA/Type 1

We have introduced in Informatik III languages of type 0 (arbitrary grammars) and of type 1 (contextsensitive grammars).

What exactly is the difference?

- We have shown in Informatik III that **type 0** languages are exactly the r.e. languages (i.e. **acceptable by DTM's**).
- We show in this section, that **type 1** languages are a strict subset of the decidable (recursive) languages. We also provide a machine model that fits for this class: the linear bounded automaton (LBA).

contextsensitive (cs) **iff** $\forall P \to Q \in R$:

- **either** $\Big(\exists u, v, \alpha \in (V \cup T)^* \; \exists A \in V \; \big(P = uAv$ and $Q = u\alpha v$ with $|\alpha| \geqslant 1\big)\Big)$, **or** the rule has the form $S \to \varepsilon$
- S does not occur in any conclusion of a rule.

That means:

- A variable A is transformed into a string α of length at least 1. The word does not get shorter, unless $\varepsilon \in L$.
- The replacement of A by α is only done, if the **context** left and right, is there.

A language that is not of type 2

Example 1.57 (Powers of 2)

We consider the following language

$$L_{\text{powers of } 2} := \{a^{2^i} \mid i \in \mathbb{N}\}$$

This language is not contextfree (pumping lemma for CFL's).

How to define a contextsensitive grammar for $L_{\text{powers of } 2}$?

Consider the following grammar G:

$$
\begin{array}{llcl}
1. & S & \rightarrow & ACaB \\
2. & Ca & \rightarrow & aaC \\
3. & CB & \rightarrow & DB \\
4. & CB & \rightarrow & \mathbf{E} \\
5. & aD & \rightarrow & Da \\
6. & AD & \rightarrow & AC \\
7. & \mathbf{aE} & \rightarrow & \mathbf{Ea} \\
8. & \mathbf{AE} & \rightarrow & \varepsilon
\end{array}
$$

- A, B are endmarkers. C moves through between them and doubles the number of a's.
- When C hits the B, it becomes D or $\mathbf{E}$.
- In the case of D, D goes left and introduces a C when the start is hit: the process starts again.
- In the case of $\mathbf{E}$, it also goes left, leaves the a's unchanged and leads to a terminal word: a string of a's.
- Show by induction on the number of steps, that the following holds (if 4. is never used): all generated strings are of the form
  - S,
  - Aa^iCa^jB, where $i + 2j$ is a power of 2,
  - Aa^iDa^jB, where $i + j$ is a power of 2.

But G is not contextsensitive (or is it)?

To create a contextsensitive grammar G', we introduce composite variables and rewrite the rules of grammar G:

transform $w = AaaaaCaaaB$ into $\mathbf{w}' = [\mathbf{Aa}]\mathbf{aaa}[\mathbf{Ca}]\mathbf{a}[\mathbf{aB}]$.

We need the following new composite variables: $[ACaB]$, $[Aa]$, $[ACa]$, $[ADa]$, $[AEa]$, $[Ca]$, $[Da]$, $[Ea]$, $[aCB]$, $[CaB]$, $[aDB]$, $[aE]$, $[DaB]$, $[aB]$.
Rules 1, 3, 4, 6, and 8 remain unchanged:

$$\begin{array}{llcl} 1'. & S & \rightarrow & [ACaB] \\ 3'. & [aCB] & \rightarrow & [aDB] \\ 4'. & [aCB] & \rightarrow & [aE] \\ 6'. & [ADa] & \rightarrow & [ACa] \\ 8'. & [AEa] & \rightarrow & a \end{array}$$

Rules 2, 5, and 7 are as follows:

$$
\begin{array}{lrcl}
2'. & [Ca]a & \rightarrow & aa[Ca] \\
 & [Ca][aB] & \rightarrow & aa[CaB] \\
 & [ACa]a & \rightarrow & [Aa]a[Ca] \\
 & [ACa][aB] & \rightarrow & [Aa]a[CaB] \\
 & [ACaB] & \rightarrow & [Aa][aCB] \\
 & [CaB] & \rightarrow & a[aCB] \\
5'. & a[Da] & \rightarrow & [Da]a \\
 & [aDB] & \rightarrow & [DaB] \\
 & [Aa][Da] & \rightarrow & [ADa]a \\
 & a[DaB] & \rightarrow & [Da][aB] \\
 & [Aa][DaB] & \rightarrow & [ADa][aB] \\
7'. & a[Ea] & \rightarrow & [Ea]a \\
 & [aE] & \rightarrow & [Ea] \\
 & [Aa][Ea] & \rightarrow & [AEa]a
\end{array}
$$

Then: $S \Longrightarrow_G^* w$ **if and only if** $S \Longrightarrow_{G'}^* w'$.

- Given a language L and a grammar G with $L = L(G)$.
- How does a DTM accept L?
- It just simulates a derivation in G:
  - Given an input w,
  - a NTM starts with S and guesses an arbitrary derivation.
  - If it has derived a terminal word u, this is compared with w. If $u = w$, then it is accepted.

We repeat the correspondence of type 0 and DTM's from Informatics III.

Theorem 1.58 (DTM corresponds to type 0)

Let L be of type 0. Then there exists a DTM *which accepts L.*

Proof

It suffices to construct a NTM, because it can be simulated by a DTM.

- Let L be a language over T of type 0.
- Then there is $G_0 = (V, T, R, S)$, generating L.
- We construct a indeterministic 2-DTM M_0 which accepts L: It guesses a derivation G_0.
- M_0 reads input w on the first tape.
- It generates on tape 2 the start symbol S of the grammar.

$$s_0, \begin{array}{l} \#w\underline{\#} \\ \underline{\#} \end{array} \vdash^*_{\mathrm{M}_0} s_0, \begin{array}{l} \#w\underline{\#} \\ \#\underline{S} \end{array}$$

Proof (continued)

- Then M_0 guesses on tape 2 a derivation in G_0. For $S \Longrightarrow^* u \Longrightarrow^* \ldots$ it computes as follows

$$s_0, \begin{matrix} \#w\underline{\#} \\ \#\underline{S} \end{matrix} \vdash^* \begin{matrix} \#w\underline{\#} \\ \#u\underline{\#} \end{matrix}$$

- M_0 guesses indet. a rule $P \rightarrow Q \in R$.
- It searches in the string on tape 2 indeterministically for P.
- It replaces this P by Q.
- If Q is longer or shorter than P, M_0 uses the shift machines S_R and S_L to adjust the rest of the word accordingly.
- M_0 chooses the next rule.

Proof (continued)

- **M_0 guesses indeterministically to compare the word on tape 2 with w.**
  If $u = w$, then M_0 halts, otherwise M_0 does never stop.
  Then:

M_0 halts at input w

iff

There is a derivation $S \Longrightarrow^*_{G_0} w$

(which can be simulated by M_0 on tape 2)

iff

$w \in L(G_0)$.

Theorem 1.59 (DTM languages are of type 0)

Let L be a language that is accepted by a DTM. Then L is of type 0, i.e. L is generated by a grammar G.

Proof

Given L and DTM $\mathcal{M}$ accepting L.

We construct G generating L:

- It generates an arbitrary word $A_1 \dots A_n$,
- and simulates $\mathcal{M}$ on Input $a_1 \dots a_n$.
- If w is accepted from $\mathcal{M}$, then G transforms $A_1 \dots A_n$ into $a_1 \dots a_n$.
- If $a_1 \dots a_n$ is not accepted by $\mathcal{M}$, then G does never transform $A_1 \dots A_n$ into terminals.

Proof (continued)

Turing machines in a **special format**:

- Normally for input va the start configuration is

$$s, \#va\underline{\#}$$

- Our DTM's use the following start configuration for input va:

$$\Subset v\underline{a} \Supset$$

Proof (continued)

- During the whole computation:
  - the left end is marked with $\Subset$ und
  - the right end is marked with $\Supset$.
- If it needs more space to the right, it should move the $\Supset$.
- **For each "normal" DTM M accepting a language L, there is a M' in the new format.**

Example 1.60 (TM with endmarkers)

The following TM M accepts $\{a^n \mid n \in \mathbb{N}\}$:

- Let $M = (\{q_1\}, \{a, b\}, \delta, q_1)$
- with δ:

$$
\begin{aligned}
(q_1, a) &\mapsto q_1, L \qquad \mapsto \\
(q_1, b) &\mapsto q_1, b \\
(q_1, \Subset) &\mapsto h, \Subset
\end{aligned}
$$

The grammar G works for M as follows:

- G guesses a word. G also codes the state and the head position of M.
- G starts as follows:

$$S \Longrightarrow_G^* \Subset \begin{matrix} w \\ w \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \Supset .$$

How to describe this with rules?

- The following rules suffice to produce (any) input:

$$S \rightarrow \Subset A_1 \mid \Subset \left\langle \begin{matrix} \# \\ \# \\ s_0 \end{matrix} \right\rangle \Supset$$

$$A_1 \rightarrow {}^{c}_{c} A_1 \mid \left\langle \begin{matrix} c \\ c \\ s_0 \end{matrix} \right\rangle \Supset \qquad \forall c \in \{a, b\}$$

The second rule in the first line is for input ϵ.

- The two and three level symbols are variables of G!
  State and head position are coded in the third
- Example:

$$S \Longrightarrow_G^* \Subset \begin{matrix} a\,a \\ a\,a \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \Supset .$$

If $(q,a)\Delta(q',a')$: $\forall b \in T \cup \{\#\}$:

$\rightsquigarrow$ **blackboard 1.10.**

If $(q,a)\Delta(q',R)$: $\forall b,c,d \in T \cup \{\#\}$:

$$\left\langle \begin{matrix} b \\ a \\ q \end{matrix} \right\rangle \begin{matrix} d \\ c \end{matrix} \rightarrow \begin{matrix} b \\ a \end{matrix} \left\langle \begin{matrix} d \\ c \\ q' \end{matrix} \right\rangle \text{ and } \left\langle \begin{matrix} b \\ a \\ q \end{matrix} \right\rangle \Supset \rightarrow \begin{matrix} b \\ a \end{matrix} \left\langle \begin{matrix} \# \\ \# \\ q' \end{matrix} \right\rangle \Supset$$

If $(q,a)\Delta(q',L)$: $\forall b,c,d \in T \cup \{\#\}$:

$$\begin{matrix} d \\ c \end{matrix} \left\langle \begin{matrix} b \\ a \\ q \end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix} d \\ c \\ q' \end{matrix} \right\rangle \begin{matrix} b \\ a \end{matrix} \text{ and } \Subset \left\langle \begin{matrix} b \\ aq \end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix} \Supset \\ q' \end{matrix} \right\rangle \begin{matrix} b \\ a \end{matrix}$$

If $(q,\Subset)\Delta(q',R)$: $\forall c,d \in T \cup \{\#\}$: $\left\langle \begin{matrix} \Subset \\ q \end{matrix} \right\rangle \begin{matrix} d \\ c \end{matrix} \rightarrow \Subset \left\langle \begin{matrix} d \\ c \\ q' \end{matrix} \right\rangle$.

Note that all these symbols are variables!

If M halts, then the input is some element from V^*. Then we do the following:

- We clear the lower track.
- We only keep the upper track, the **original input**.
- Blanks and $\Subset$, $\Supset$ have to be removed.
- This has to be done with appropriate rules.

In general, a grammar is not bounded:

- The simulated DTM may move the $\Supset$ to the right or left as needed.
- So we have a derivation

$$S \Longrightarrow^*_G \Subset \begin{matrix} u \\ u \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \Supset \Longrightarrow^*_G \Subset \begin{matrix} u_1 \\ x_1 \end{matrix} \left\langle \begin{matrix} c \\ b \\ h \end{matrix} \right\rangle \begin{matrix} u_2\# \ldots \# \\ x_2\# \ldots \# \end{matrix} \Supset$$

- **The additional blanks have to be removed, as well as the $\Supset$ and the $\Subset$ (the word gets shorter)!** We need ε productions for that: $\Subset \rightarrow \epsilon$, $\Supset \rightarrow \epsilon$, $\# \rightarrow \epsilon$.
- What remains is just the word w, the original input.

Thus G generates w iff the DTM M accepts w.

Theorem 1.58 and Theorem 1.59 taken together can also be reformulated as follows:

Theorem 1.61 (r.e. corresponds to DTM's)

A language is recursively enumerable (r.e.) *if and only if it as accepted by a* DTM.

The importance of the construction in the proof of Theorem 1.59 is that **almost the same construction** works for LBA'a and contextsensitive grammars.

Definition 1.62 (Linear bounded automaton (LBA))

A *linear bounded automaton (LBA)* is an indeterministic Turing machine (NTM) M which satisfies:

- Initial configuration of M:

$$\begin{cases} s_0, \Subset w\underline{a} \Supset, & \text{for input } wa \neq \epsilon; \\ s_0, \Subset \underline{\#} \Supset, & \text{for input } \epsilon. \end{cases}$$

- If M reads $\Subset$, it makes a step to the right.
- If M reads $\Supset$, it makes a step to the left.
- M never writes $\Subset$ or $\Supset$.

Consider again Example 1.57.**How to define a LBA that decides $L_{\text{powers of 2}}$?**

Use an LBA with 4 tapes (or tracks).

Theorem 1.63 (Type 1 languages correspond to LBA'a)

L is contextsensitive iff L is accepted by a LBA.

Proof

LBA'a accept languages of type 1:

- **LBA with 2 tracks**. Upper track: Input, lower track: simulating a derivation of G.
- G ist **bounded**: the derived word does not get shorter.
- It suffices to consider derivations of length up to the length of the input.
- So the space between $\Subset$ and $\Supset$ suffices.

Proof (continued)

LBA'a accept languages of type 1:

- We build on (and modify) the proof of Theorem 1.58.
- The simulated **DTM** got the input between $\Subset$ and $\Supset$ but the $\Supset$ could be moved (to enlarge the storage).
- Now we construct a **bounded** grammar, which simulates the computation of the LBA M.
- **Problem 1**: rules $\left\langle \begin{array}{c} b \\ a \\ q \end{array} \right\rangle \Supset \rightarrow \begin{array}{c} b \\ a \end{array} \left\langle \begin{array}{c} \# \\ \# \\ q' \end{array} \right\rangle \Supset$.
- **Problem 2**: rules $\Subset \rightarrow \epsilon$, $\Supset \rightarrow \epsilon$, $\# \rightarrow \epsilon$.

Proof (continued)

Solution: Hide $\Supset, \Subset$ within the variables by introducing new 4- and 5-level symbols.
Guessing the word w is now done by the rules (compare with Slide 138):

$$S \rightarrow \left\langle \begin{matrix} \Subset \\ a \\ a \end{matrix} \right\rangle A_1 \mid \left\langle \begin{matrix} \Subset \\ \Supset \\ a \\ a \\ s_0 \end{matrix} \right\rangle$$

$$A_1 \rightarrow {}^{a}_{a} A_1 \mid \left\langle \begin{matrix} \Supset \\ a \\ a \\ s_0 \end{matrix} \right\rangle \Supset \qquad \forall a \in T$$

$\rightsquigarrow$ **blackboard 1.11**

Lemma 1.64 ($Comp(\mathrm{M})$ is of Type 1)

Let M be a Turing machine (indeterministic or not). Then the following holds:

1. $Comp(\mathrm{M})$ *is contextsensitive.*
2. $Comp_{halt}(\mathrm{M})$ *is contextsensitive.*

Corresponding LBA'a as well as cs grammars can be efficiently constructed.

Compare this Lemma with Theorem 1.39.

Proof

Given an arbitrary TM M, we construct an LBA B as follows. B gets a huge string $\#c_1\#c_2\#\ldots\#$ and has to decide whether this is an accepting sequence of configurations of a word w for the DTM M.
It is easy to verify that this can be done (just moving the head on the input): zig-zagging between c_i and c_{i+1} to verify that c_{i+1} is a legal successor configuration.
By Theorem 1.63 we can construct a cs grammar as well.

Corollary 1.65 (Emptiness Problem for LBA)

The problem to determine for a given LBA M *whether* $L(M) = \emptyset$ *is* **undecidable**.
The problem to decide for a given contextsensitive grammar G *whether* $L(G) = \emptyset$ *is undecidable.*

Lemma 1.66 (LBA'a as deciders)

The following problems can be decided by an LBA:

1. *Does a dfa M accept a word w?*
2. *Does a cf grammar generate a word w?*
3. *Given a dfa M, is $L(M) = \emptyset$?*

Proof: $\rightsquigarrow$ **exercise**

Theorem 1.67 (Type 1 languages are recursive)

A contextsensitive language is **recursive**. *For each contextsensitive grammar G, one can construct a DTM M which solves the word problem for G.*
In other words: the **word problem for LBA'a is decidable**.

Proof

There are two possibilities.

Grammars: To check a given word w of length n for derivability in the grammar, it suffices to check **all possible words up to length n** (because the grammar is bounded). These are finitely many.

LBA: If a LBA does not terminate, it has to repeatedly go into the same configuration. But there are only finitely many configurations, so that can be checked.

Theorem 1.68 (Diagonalisation)

Let $\mathrm{M}_1, \mathrm{M}_2, \ldots, \mathrm{M}_i, \ldots$ an **enumeration** *of DTM's which decide certain languages $L(\mathrm{M}_i)$ (over an alphabet containing $\{0, 1\}$).*
Then there is a **recursive** *language L_{rec} over $\{0, 1\}$ which is* **not decided** *by any of the machines.*

Proof

Let L_{rec} over $\{0,1\}$ defined as follows (val(w) is the value of w written in binary):

$w \in L_{rec}$ **iff** $\mathrm{M}_{\mathrm{val(w)}}$ prints “N” for input w

If L_{rec} would be decided by some M_k ($k \in \mathbb{N}$), then one consider w with val$(w) = k$:

M_k prints “Y” for input w
iff
$w \in L_{rec}$
iff
$\mathrm{M}_{\mathrm{val}(w)} = \mathrm{M}_k$ prints “N” for input w, Contr.!!.

Theorem 1.69 (Type 1 strictly contained in Rec)

There is a recursive language which is not contextsensitive.

Proof

We have to show that all DTM's that decide languages of type 1 can be enumerated. Obviously, all type 1 languages over $\{0, 1\}$ can be represented: gödelisation of grammars.
Thus the contextsensitive grammars are enumerable.
With Theorem 1.67 the respective DTM's can be effectively constructed.

A more natural example is the set of true formulae in Presburger arithmetic. It is recursive, but any decision procedure is at least double exponential (so it can not be decided by a LBA (see Section 3).

Lemma 1.70 (Closure properties of Type 1)

The contextsensitive languages are closed under union, intersection, inverse homomorphisms, concatenation, ε-free homomorphisms, and Kleene closure.

Proof.

This is best done with LBA'a: $\rightsquigarrow$ **exercise**. □

Lemma 1.71 (Type 1 not closed under hom)

The contextsensitive languages are not closed under homomorphisms. *The same is true for the class of* recursive *languages.*

Proof.

We consider $Comp(\mathrm{M})$ for a TM M. In Lemma 1.64, we have shown that this set is contextsensitive. We modify this set slightly, by putting a copy of the first ID in front of each string. This copy is using a different alphabet Σ' which is isomorphic to Σ. So we consider the set

$$\{w_1' w_1 \# w_2^R \# w_3 \# w_4^R \# \ldots \# w_n \,|\, w_1 \# w_2^R \# w_3 \# w_4^R \# \ldots \# w_n \in Comp(\mathrm{M})\}$$

We define a homomorphism on $\Sigma' \cup \Sigma$ ($\rightsquigarrow$ **exercise**) such that $h(Comp(\mathrm{M}))$ is the set

$$\{w_1' h \,|\, \mathrm{M} \text{ terminates when started with } w_1\}$$
$$\cup \quad \{w_1' \,|\, w_1' \text{ is an initial ID of } \mathrm{M}\}$$

This set is not contextsensitive, not even recursive (**the halting problem is reducible to it**). □

- We have just seen, that the class of *recursive* languages, *Rec*, is inbetween the *contextsensitive* and the *recursively enumerable* classes.
- What can we say about *closure properties* of **Rec**?

The following theorem was open for several decades. It was proved independently at the end of the 80'ies by Neil Immerman and Róbert Szelepcsényi.

Theorem 1.72 (Type 1 closed under complements)

Contextsensitive languages are closed under complements.

The proof follows from a more general theorem later (Theorem 4.4 on Slide 533).

1.6 re = hom(cs)

What is the relation between type 1 and type 0 languages?

Theorem 1.73 (Each r.e. L is of the form h(type1))

Each type 0 language is the homomorphic image of a contextsensitive language. *I.e. each r.e. language has the form $h(L)$ where h is a homomorphism and L is a language of type 1.*

Proof

Let L be r.e. and M a DTM that accepts L. Let c be a new symbol not in Σ and

$$L' = \{wc^i | \text{ M accepts } w \text{ by a sequence of moves in which the head never moves more than } i \text{ positions to the right of } w\}$$

- Obviously, L' is accepted by a LBA (it simulates M, treats c as blank and halts when it goes beyond the sequence of c's).
- Define h as follows: $h(a) = a$ for all $a \in \Sigma$, $h(c) = \epsilon$. Then $L = h(L')$.

1.7 Ehrenfeucht

- For a given language L, we call two morphisms **equivalent**, if they agree on L.
- **Is there a smaller set $T \subset L$ so that checking L-equivalence can be reduced to just check T?**
- Such a T is called **test set**.

Ehrenfeucht Hypothesis for languages L

For each language $L \subseteq \Sigma^*$ **there is a finite test set T**.

An interesting question is, whether such a set **can be effectively constructed**.

We have defined the notion of a (homo-) morphism f from Σ^* in Γ^*. An interesting notion is to define the equivalence of two morphisms with respect to a language L.

Definition 1.74 (f, g equivalent wrt. $L \subseteq \Sigma^*$, $f =_L g$)

Two morphisms $f, g : \Sigma^* \rightarrow \Gamma^*$ are equivalent wrt. $L \subseteq \Sigma^*$, written $f =_L g$, if for all $w \in L$: $f(w) = g(w)$.

Given two morphisms f, g. Is it decidable whether they are equivalent wrt. Σ^*?

Example 1.75

Let $\Sigma = \Gamma = \{a, b\}$ and let

$$\begin{aligned} f(a) &:= a \\ f(b) &:= bb \\ g(a) &:= abb \\ g(b) &:= b \end{aligned}$$

Then we have $f \neq g$ but

$$f =_{(abb)^*} g.$$

Lemma 1.76

$f =_L g$ implies $f(L) = g(L)$ but the converse does not hold.

Lemma 1.77 (Undecidability of $f(L) = g(L)$)

The following problem is undecidable. Given a cf language L and two morphisms f, g, decide whether $f(L) = g(L)$.

Proof.

Reduction of equality of contextfree languages.

$\rightsquigarrow$ **blackboard 1.12**.

Lemma 1.78 (Undecidability of $f =_L g$)

The following problem is **undecidable**. **Given** *a language L of type 1 and two morphisms f, g,* **decide** *whether $f =_L g$.*

Proof.

Reduction of PCP (next chapter).
$\leadsto$ **blackboard 1.13**

Does it suffice to test the equality of morphisms on a subset of L only?

Definition 1.79 (Test-Set)

A set $T \subseteq L$ of a language $L \subseteq \Sigma^*$ is called **Test-Set**, if for all morphisms f, g the following holds:

$$f =_L g \text{ if and only if } f =_T g$$

In Example 1.75, the set $\{abb\}$ is a test set. Our first result is to determine a test set for for regular languages. While the proof is elementary, it is a bit complicated and makes use of the following lemma.

Lemma 1.80 (Solving the equation $xy = yz$)

Let $V = \{x, y, z\}$ a set of variables and Σ an alphabet. We are interested in determining morphisms $h : V^* \to \Sigma^*$ such that $h(xy) = h(yz)$: such an h is called a **solution** *of the equation $xy = yz$. For $h(x), h(z) \neq \varepsilon$ the following are equivalent:*

- *h is a* **solution** *of the equation $xy = yz$.*
- *There are $r, s \in \Sigma^*$ and $k \in \mathbb{N}_0$ with*
  - $h(x) = rs$, $h(y) = (rs)^k r$, $h(z) = sr$.

Proof.

$\rightsquigarrow$ **exercise**

Theorem 1.81 (Test set for regular languages)

Let L be a regular language given by a deterministic finite automaton $A = (K, \Sigma, \delta, s_0, F)$. Then the following set is a finite test set for L:

$$\{w : |w| \leqslant 2|K|\}$$

Proof.

Let $f, g : \Sigma^* \to \Gamma^*$ and $f =_L g$. We assume $w = a_1 \ldots a_n \in L$ a word of **minimal** length s.t. $f(w) \neq g(w)$ and $n \gneqq 2|K|$. □

Proof (cont.)

As $w \in L$, there must exist a state q which is visited at least three times. Therefore we can split w into four parts $w = u_1 u_2 u_3 u_4$:

$$\delta(q_0, u_1) = q = \delta(q, u_2) = \delta(q, u_3)$$

and $\delta(q, u_4) \in F$. Obviously the three words $u_1 u_2 u_4$, $u_1 u_3 u_4$, $u_1 u_4$ are also all in L. We also have $u_2 \neq \epsilon \neq u_3$ which means that f and g coincide on these three words.
In the following we will use these identities:

$$\begin{array}{lcl} f(u_1)f(u_2)f(u_4) & = & g(u_1)g(u_2)g(u_4) \\ f(u_1)f(u_3)f(u_4) & = & g(u_1)g(u_3)g(u_4) \\ f(u_1)f(u_4) & = & g(u_1)g(u_4) \end{array}$$

We compare $f(u_1)$ with $g(u_1)$.
We get 3 cases: $|f(u_1| = |g(u_1|$, $|f(u_1| \gneq |g(u_1|$, $|f(u_1| \lneq |g(u_1|$.

Proof (cont.)

Case 1: $|f(u_1| = |g(u_1|$**.** Then, using the identities, we have $f(u_1) = g(u_1)$, $f(u_4) = g(u_4)$, $f(u_3) = g(u_3)$, and thus

$$f(w) = f(u_1u_2u_3u_4) = f(u_1)f(u_2)f(u_3)f(u_4) = g(u_1)g(u_2)g(u_3)g(u_4) = g(u_1u_2u_3u_4) = g(w)$$

which is a contradiction

Case 2: and **Case 3:** are similar. It suffices to do one of them.
In case 2 ($|f(u_1| \gneqq |g(u_1|$), we have $f(u_1) = g(u_1)x$ and $g(u_4) = xf(u_4)$ for a $x \in \Gamma^*$. Then we have $g(u_1)xf(u_2)f(u_4) = g(u_1)g(u_2)xf(u_4)$.
Therefore $xf(u_2) = g(u_2)x$ or: $g(u_2)x = xf(u_2)$.
In the same way, we can show $xf(u_3) = g(u_3)x$ or: $g(u_3)x = xf(u_3)$.
We also have $f(u_2) \neq \epsilon \neq g(u_2)$: were $f(u_2) = \epsilon$, then $\epsilon = g(u_2)$ and thus $f(w) = \ldots = g(w)$, a contradiction.
We are now in a position to apply Lemma 1.80 (twice, for each equation). Putting this together and computing $f(w)$ and $g(w)$ gives us the contradiction $f(w) = g(w)$ ($\rightsquigarrow$ **blackboard 1.14**).

Can we effectively construct test sets for regular languages?

Corollary 1.82 (Test set for regular languages)

The problem to decide for a given regular expression ex and morphisms f, g whether $f =_L g$ is decidable.

Proof.

Obvious. □

We note without proof: (Albert/Culik 1980)

Theorem 1.83 (Test set for contextfree languages)

- *For each contextfree language L one can* effectively construct a finite test set.
- *The problem to* decide for a given contextfree language L *and morphisms f, g whether* $f =_L g$ is decidable.

It would be nice to **always guarantee** a finite test set: this was stated as a hypothesis by Ehrenfeucht.

Theorem 1.84 (Lawrence, Albert, Guba 1985)

For each language $L \subseteq \Sigma^*$ there exists a finite test set $T \subset L$.

Before turning to the proof, we note a few simple consequences.

Corollary 1.85 (Decidability for fixed L)

Let $L \subseteq \Sigma^*$ be given. Then the problem to decide *for given morphisms f, g* whether $f =_L g$ *is* decidable.

Another interesting implication is by considering Lemma 1.78.

Corollary 1.86 (Test sets for Type 1 are not computable)

The problem to compute for a given contextsensitive language a finite test set is not algorithmically solvable.

Note that here we consider the **language as a parameter**: it is part of the input (represented as a contextsensitive grammar (which is a finite object)).

Definition 1.87 (Ehrenfeucht for $\mathbb{Z}[x_1, \ldots, x_r]$)

Let $\mathbb{Z}[x_1, \ldots, x_r]$ be the ring of polynomials in $x_1, \ldots, x_r$.
The Ehrenfeucht hypothesis for $\mathbb{Z}[x_1, \ldots, x_r]$ is the following. For all $Q \subseteq \mathbb{Z}[x_1, \ldots, x_r]$ there is a finite $Q' \subseteq Q$ such that for all $\vec{z} \in \mathbb{Z}^r$ the following holds

$$(\forall q \in Q : \; q(\vec{z}) = 0) \; \underline{\textbf{iff}} \; (\forall q' \in Q' : \; q'(\vec{z}) = 0)$$

Definition 1.88 (Ehrenfeucht for word equations)

Let $V = \{v_1, \ldots, v_r\}$ be a set of r variables. The Ehrenfeucht hypothesis for V is the following. For all $S \subseteq V^* \times V^*$ there is a finite $T \subseteq S$ such that for all morphisms $h : V \to \Sigma^*$ the following holds

h is solution of S **iff** h is solution of T

The set S constitutes a set of equations. Simple example is $\langle xy, yz \rangle$, i.e. the equation considered in Lemma 1.80.

Lemma 1.89 (Word equations vs standard hypothesis)

The Ehrenfeucht hypothesis for word equations implies the standard hypothesis.

Proof.

For $\Sigma = \{a_1, \ldots, a_r\}$ we define copies $\hat{\Sigma}$ and $\tilde{\Sigma}$ and natural embeddings $\Psi_1 : \Sigma^* \to \tilde{\Sigma}^*$, $\Psi_2 : \Sigma^* \to \hat{\Sigma}^*$. Let

$$S := \{\langle \Psi_1(w), \Psi_2(w) \rangle \mid w \in L\}$$

be a (possibly infinite) system of word equations. By assumption, there is an equivalent finite set S' involving only finitely many words $\{w_1, \ldots w_k\} =: T$. **We claim T is a test set for L.** $\rightsquigarrow$ **blackboard 1.15** □

Lemma 1.90 ($\mathbb{Z}[x_1, \ldots, x_r]$ and word equations)

The Ehrenfeucht hypothesis for $\mathbb{Z}[x_1, \ldots, x_r]$ implies the Ehrenfeucht hypothesis for word equations.

Proof (Sketch).

How do we encode a word equation in the polynomial ring? We code each single word as an m-ary object, but we also have to make sure that each position in the whole word is unique:

- a word $a_0 a_1 \ldots a_{n-1} \in \Sigma^n$ of length n is written in the polynomial ring as $pol(w) = a_0 + a_1 X^1 + \ldots + a_{n-1} X^{n-1}$, and
- a word $w_1 \ldots w_m \in \Sigma^*$ is written as $pol(w_1 \ldots w_m) = pol(w_1) + pol(w_2) X^{|w_1|} + \ldots + +pol(w_m) X^{|w_1 w_2 \ldots w_{m-1}|}$.

In that way, we assign each h and S a set Q of polynomials s.t. h is a solution of S iff $(\forall q \in Q : q(\vec{z}) = 0)$ (for any $\vec{z} \in \mathbb{Z}^r$). So a finite Q_{fin} induces a finite S_{fin}. □

- So we are left with an algebraic problem.
- Consider $Q \subseteq \mathbb{Z}[x_1, \ldots, x_r]$. Can we show that each such Q has a finite base?
- Then the Ehrenfeucht hypothesis for $\mathbb{Z}[x_1, \ldots, x_r]$ follows (exercise).

1. It suffices to show the existence of finite bases for ideals (invented by Kummer as an analogue for prime numbers).
2. A ring R with the property that each ideal in R has a finite base is called noetherian.
3. $\mathbb{Z}$ is a noetherian ring with 1.
4. **Hilbert's Basisssatz:** If a ring R with 1 is noetherian, then so is $R[x]$ (famous theorem of Hilbert). And so is $R[x_1, \ldots, x_r]$.

Definition 1.91 (Ideal $\mathcal{I}$, noetherian ring)

A subset $\mathcal{I}$ of a commutative ring R with 1 is called ideal if it satisfies the following:

1. $0 \in \mathcal{I}$,
2. if $a, b \in \mathcal{I}$ then $a - b \in \mathcal{I}$,
3. if $a \in \mathcal{I}$ and $r \in R$ then $ra \in \mathcal{I}$.

If all ideals of R have a finite base, the ring is called noetherian.

For our purpose, the ring R we need is $\mathbb{Z}$. A typical ideal is the set of all even numbers $2\mathbb{Z}$.

Ideals date back to Kummer $\rightsquigarrow$ **blackboard 1.16**

Theorem 1.92 (Hilbert)

If R is noetherian, then so is $R[x_1, \ldots, x_n]$.

Proof.

1. It suffices to prove the result for $n = 1$.
2. The main idea is to relate ideals in $R[x]$ to ideals in R: by using the coefficients of the highest powers of the polynomials.

Let $\mathcal{I}$ be an ideal in $R[x]$. **We have to show that there is a finite basis.**

Proof (cont.)

The set of coefficients of the highest powers:

$$\mathcal{I}' = \{a : \ ax^k + bx^{k-1} + \ldots \in \mathcal{I}\}\} \cup \{0\}$$

forms an ideal (why?): by assumption, it is finitely generated by a set $\{a_1, \ldots, a_r\}$.

We denote by $pol(a_i) = a_i x^{n_i} + \ldots$ the polynomial for a_i. **Let N be the maximum of the n_i.**

Let $Base_N = \{pol(a_1), \ldots, pol(a_r)\}$. Our final base will be a finite union $Base_N \cup \ldots \cup Base_0$.

(1) All polynomials in $\mathcal{I}$ of degree $\geqslant N$ can already be obtained from $Base_N$.
$\rightsquigarrow$ **blackboard 1.17**

Proof (cont.)

(2) The coefficients of x^{N-1} in all polynomials in $\mathcal{I}$ (plus 0) form an ideal in R (why?) **So there is a finite base $\{a_{r+1}, \ldots, a_s\}$ of it.**

(3) Let $Base_{N-1} = \{pol(a_{r+1}), \ldots, pol(a_s)\}$. All polynomials of degree $N-1$ can be written as a linear combination of these base functions and functions of degree at most $N-2$: see (1).

(4) We then consider the coefficients of $x^{N-2}, \ldots, x^0$ and add all the corresponding polynomials which gives us $Base_{N-2}, \ldots, Base_0$: the union constitutes the final base.

1.8 Lindenmayer

As opposed to the grammars in the Chomsky hierarchy, for Lindenmayer systems the following holds:

- Production rules are applied in parallel, not sequentially.
- There is no distinction between terminals and non-terminals.

`http://www.jjam.de/Java/Applets/Fraktale/Lindenmayer.html`

- Aristid Lindenmayer (1968): **Mathematical models for cellular interaction in development, I and II;**, **J. Theoret. Biol.** 18, 280-315.
- Modelling the development of simple organisms.

Definition 1.93 (Lindenmayer System G, (0L, D0L))

A Lindenmayer system is a tuple $G = (\Sigma, \mathcal{R}, w)$ with:

1. Σ is the alphabet,
2. $\varepsilon \neq w \in \Sigma^*$,
3. $\mathcal{R}$ is a finite set of rules R of the form: $R \subset \Sigma \times \Sigma^*$. For $(P, Q) \in R$ we write $P \rightarrow_G Q$.

We assume that for each $v \in \Sigma$ there is **at least one** word w' with $v \rightarrow_G w'$ (otherwise we add $v \rightarrow_G v$). If there is for each $v \in \Sigma$ exactly one word w', $v \rightarrow_G w'$, then G is called deterministic.

A Lindenmayer system G is also called $0L$. A deterministic system is called $D0L$. The language generated by G is the set of all words that can be derived from w with all the rules in G.

A language L is called $0L$-language (resp. $D0L$-language), if there is a $0L$ system (resp. $D0L$ system) that generates L. The class of languages is denoted by $\mathbf{0L}$, resp. $\mathbf{D0L}$.

Are there $0L$ ($D = L$) grammars for the following languages?

- $L_1 = \{a^n : n \geqq 2\}$
- $L_2 = \{(ab)^{2^n} : n \geqq 0\}$

$\leadsto$ **blackboard 1.18**

Lemma 1.94

The language $\{a^n : n \geqslant 2\}$ is no $D0L$ language.

$\leadsto$ **blackboard 1.19**

Some interesting examples

- $\{\epsilon, a, a^2\} \in \mathbf{0L}$ but $\{\epsilon, a^5, a^{10}\} \notin \mathbf{0L}$,
- $\{a\}$ and $\{a^3\} \in \mathbf{0L}$ but $\{a, a^3\} \notin \mathbf{0L}$,
- $(\{a^n : n \geqq 2\} \cap \{a, a^4\}) \notin \mathbf{0L}$,
- $\{a\}$ and $\{\epsilon,^2 a\} \in \mathbf{0L}$, but
  $\{a\}\{\epsilon, a^2\} = \{a, a^3\} \notin \mathbf{0L}$,
- $\{a^n b^n : n \geqslant 1\} \notin \mathbf{0L}$,

Lemma 1.95 (Non Closedness of $0L$)

$0L$ *is* **not closed** *under*

- ε*-free homomorphisms,*
- *union,*
- *intersection with regular languages,*
- *concatenation.*

$\rightsquigarrow$ **blackboard 1.20**

Lemma 1.96 (Incomparability of $0L$)

$0L$ is incomparable *wrt. set inclusion to the following classes:*

- **finite languages,**
- **infinite regular languages,**
- **contextfree, non-regular languages.**

$\leadsto$ **blackboard 1.21**

We note without proof:

Lemma 1.97 ($0\mathrm{L} \subseteq$ CSL)

$\mathbf{0L}$ *is strictly contained in the class of contextsensitive languages.*

Does that imply, that the word problem for $0L$-languages is decidable?

Lemma 1.98 (Derivation bounded by k_G)

Let $G = (\Sigma, \mathcal{R}, w_0)$ be a $0L$ system. Then **there is a $k_G \in \mathbb{N}$** *(which can be effectively computed from G) such that for each $\varepsilon \neq w \in L(G)$ there exists a derivation*

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \ldots \Rightarrow_G w_m = w$$

with: $|w_i| \leqslant k_G|w|$ *for* $i = 0, 1, \ldots, m$.

Proof.

Let $L^i(G)$ be the set of words derivable after i steps, and

- $n := |\Sigma|$, $r := \max_{\text{rules } a \to x \in \mathcal{R}}\{|x|\}$,
- $s := \max_{x \in L^i(G), i=0,1,\ldots,n}\{|x|\}$,
- $k_G := \max\{s, r^{n+1}\}$.

$\rightsquigarrow$ **blackboard 1.22**

Corollary 1.99 (Word problem for $0\mathrm{L}$)

The word problem for $0L$ systems is decidable.

Proof.

Let $w \neq \varepsilon$ (why?). Compute the bound k_G which exists according to Lemma 1.98. We define $K_n(G, w)$ as the set of all $v \in \Sigma^*$ which satisfy (for $i = 1, \ldots m$)

$$w_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_m = v \text{ with } m \leqslant n \text{ and } |v_i| \leqslant k_G|w|$$

$K_{n+1}(G, w)$ is the union of $K_n(G, w)$ and the set

$$\{v : \exists z \in K_n(G, w) : z \Rightarrow v \text{ and } |v| \leqslant k_G|w|\}$$

We have $1 = |K_0(G, w)| \leqslant \ldots \leqslant |K_n(G, w)| \leqslant \ldots \leqslant |\Sigma|^{k_G|w|+1}$. The sequence gets stationary: there is a t such that for all $m \geqslant 1$

$$K_t(G, w) = K_{t+m}(G, w)$$

This we can reduce the check "$w \in L(G)$" to "$w \in K_t(G, w)$".

Lemma 1.100 (Undecidability of $0L$-equivalence)

It is **undecidable** *whether two $0L$ systems generate the same languages.*

Proof.

We reduce the PCP (next chapter) to this problem. Therefore let $I = \{\langle u_1, v_1\rangle_1, \langle u_2, v_2\rangle_2, \ldots, \langle u_k, v_k\rangle_k\}$ an instance of PCP over $\{a, b\}$. We define $\mathbf{0L}$ systems G_I, H_I s.t.

- $L(G_I) \supseteq \{wDu^R : w, u \in \Sigma^\star, w \neq u\}$,
- $L(H_I) = L(G_I) \cup \{wDv^R : w = u_{i_1} \ldots u_{i_t}, v = v_{i_1} \ldots v_{i_t}, t \geqslant 1, i_1, \ldots i_t \in \{1, \ldots, n\}\}$.

Obviously: **$L(G_I) \neq L(H_I)$ iff I has a solution.**

It remains to define the $\mathbf{0L}$ systems G_I, H_I:
$\leadsto$ **blackboard 1.23**

2. (Un-) Decidability

2 (Un-) Decidability
- Universal DTM
- PCP
- Tiling the Plane
- Recursive Functions
- Random Access Machines
- Hilberts 10. Problem
- Theorem of Rice
- Recursion theorem
- Oracle TMn
- Reductions

Content of this chapter:

1. Universal DTMs.
2. Post's Correspondence-problem: PCP.
3. Partial recursive functions.
4. The Grzegorczyk Hierarchie.
5. Yet another machine model: Random Access Machines. Loop-, Goto- und While-Programs.
6. Hilbert's 10. problem.
7. Theorems of Rice and Greibach.
8. smn- and Recursion- Theorem.
9. Oracle Turingmachines.
10. Reductions.

Reduction of one problem P_1 to another one P_2

We have to construct a total, computable function f, which assigns

- to each instance p_1 of P_1
- an instance p_2 of P_2, such that
- the answer to p_1 is "yes" **iff** the answer to p_2 is also "yes".

Definition 2.1 (Reduction)

Let L_1, L_2 be languages over $\Sigma = \{|\}$ (so that $\Sigma^* = \mathbb{N}$).

We say L_1 reduces to L_2 (denoted by $L_1 \leq L_2$), **iff**

there is a DTM-computable function $f : \mathbb{N} \to \mathbb{N}$, such that:

$$\forall n \in \mathbb{N} \quad \left(n \in L_1 \text{ \textbf{iff} } f(n) \in L_2\right).$$

The most important property of reductions is the following.

Lemma 2.2

If $L_1 \leq L_2$, and L_1 is undecidable*, then L_2 is also* undecidable*.*

We will introduce more reductions later in this chapter (Slides 430 and Slides 416).

Definition 2.3 (Turing machine (DTM))

A **deterministic Turing machine (DTM)** M is a tuple $M = (K, \Sigma, \delta, q_0)$, where

- K is a finite set of states $h \notin K$,
- Σ an alphabet with $L, R \notin \Sigma, \# \in \Sigma$,
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ a transition function, and
- $q_0 \in K$ the initial state.

Number of states: $|K| - 1$ (initial state is not counted). h is the **final state**.

In a δ step $\delta(q, a) = (q', x)$ a DTM can,

- depending on the current state $q \in K$ and
- depending on the letter $a \in \Sigma$, that is currently scanned,

do the following

- **move left**, if $x = L$, or
- **move right**, if $x = R$, or
- **overwrite** a, which is currently scanned, **by** $b \in \Sigma$, if $x = b \in \Sigma$.

In addition, the state is changed to $q' \in K \cup \{h\}$. As $h \notin K$, there is no transition after h has been reached.

Important:

Note that our notion of a DTM cannot print and move at the same time. Also, there is only one halting state h, not several. This is done differently in other approaches in the literature. Of course this does not change any theorems, but it affects some proofs.

The following is known from Informatics 3:

Enumeration of DTM's: There is an enumeration $M_1, M_2, \ldots$ of **all DTM's**.

Enumeration of Σ^*: There is an enumeration $w_1, w_2, \ldots$ of **all inputs**.

Universal DTM U: There is a universal machine U which simulates any DTM: $\mathsf{U}(i, j) = M_i(w_j)$. I.e. U takes as input two numbers and then starts the machine M_i on input w_j.

We now consider DTM's that are **deciders**: on each input, either Y or N is printed.

Enumeration of deciders: Is there an enumeration $M_1, M_2, \ldots$ of **all DTM's that are deciders**?

Cantor: Assume there is an enumeration. Then we define M_{new} as follows. On input w we first compute j with $w = w_j$ and then we compute $M_j(w_j)$. Finally we switch the result (from N to Y and vice versa).

There are countably many deciders $M_1, \ldots$, but they can not be enumerated by any DTM: The function that assigns to each natural number the Gödelnumber of a decider **is not computable**.

Enumeration of LBA'a: Is there an enumeration $M_1, M_2, \ldots$ of **all LBA'a**?

There is certainly an enumeration of all DTM's **that are not LBA'a**. If there were an enumeration of **all LBA'a**, then the following problem would be decidable: check for a given DTM whether it is an LBA (simply check whether it belongs to one list or the other). But this problem is undecidable (see one of the exercises).

Modified Enumeration of LBA'a as deciders: Is there an enumeration $M_1, M_2, \ldots$ of **LBA'a** that are deciders with the following property: for all contextsensitive languages, there is at least one LBA in the enumeration that accepts this language.

1. Indeed, there is such an enumeration: Enumerate all grammars, check out those that are contextsensitive and compute the respective LBA.
2. **What happens with the Cantor construction?** It simply shows us that this *enumeration can not be done by a LBA*.
3. Why is the Cantor construction not applicable to the enumeration of all DTM's? Well, it is. But we do not get a contradiction because we can not implement the switch of the outputs: DTM's simply do not have to terminate.

2.1 Universal DTM

- There is a DTM U which can **simulate arbitrary DTM's**:

  U gets as input

  - the program of a DTM M and
  - a word w on which M works.

  U **simulates** M by looking which δ transition M would do.

  Such a DTM U is called **universal Turing Machine**.

Question: What is the best format of the rules of a DTM M to feed them to a universal DTM?

Or: What do we need to describe a DTM completely?

- its alphabet,
- its states,
- its δ function and
- its starting state.

We now standardize the alphabet, the states and the start state for all DTM's. Then it suffices to just state the δ-transitions. Therefore:

- We consider an infinite $\Sigma_\infty = \{a_0, a_1, \dots\}$, such that the alphabet of each DTM can be seen as a subset of Σ_∞.
- The names of the states do not matter. We assume they are denoted by $q_1, \dots, q_n$, where the n can differ from DTM to DTM.
- We also denote by q_1 the starting state and by q_0 the halting state.

We have achieved the following:
A DTM is completely described by listing its δ transitions.

How can a description look like?

The DTM $L_{\#}$ has the rules

$$
\begin{array}{ll}
q_1, \# \mapsto q_2, L & q_2, \# \mapsto q_0, \# \\
q_1, | \mapsto q_2, L & q_2, | \mapsto q_2, L
\end{array}
$$

Assume $\#$ is the sign a_0, and $|$ is a_1. We can describe DTM $L_{\#}$ as follows:

	a_0	a_1
q_1	q_2, L	q_2, L
q_2	q_0, a_0	q_2, L

or shorter:

$$
\begin{array}{l}
Z\ S\ 2\lambda\ S\ 2\lambda \\
Z\ S\ 0a0\ S\ 2\lambda
\end{array}
$$

We use

- Z for “next line”
- S for “next column”
- λ for “left”,
- ρ for “right”,
- a for the the sign a,
- the number n for the n-th state and the n-th sign of Σ_∞.

We write n as a sequence of n strokes $|| \ldots |$.
We need a to distinguish the state from the sign that is printed: $|||a||$ stands for q_3a_2 (without a we could not distinguish q_2a_3 from q_3a_2).

So we have described the DTM by a single word:

$$ZS||\lambda S||\lambda ZS0a0S||\lambda$$

Example 2.4 (Gödelisation)

We denote by **Gödelisation** any fixed procedure to assign each DTM a natural number or a word (**Gödelnumber**, **Gödelword**) in such a way, that the DTM can be effectively computed given the number (or the word).

We give a detailed construction of a universal Turing-Machine U.

- U is a 3-DTM.
- U gets the input on 2 tapes:
  - Tape 2 U contains the Gödelization $g(\mathrm{M})$ of a DTM M.
  - Tape 1 of U contains the Gödelisation $g(w)$ of the input w for M.
- During the computation the tapes are used as follows:
  - Tape 1 contains the (contents of the) tape of M as a Gödelword.
  - Tape 2 contains the Gödelword of M.
  - Tape 3 contains the Gödelword of the actual state of M.

During the simulation the following holds for all $w \in (\Sigma_\infty \backslash \{\#\})^*$:

iff M computes $s_M, \#w\underline{\#} \vdash_M^* h, u\underline{a_i}v$
iff U computes

$$s_U, \quad \# g(\#w\#) \underline{\#}, \quad \# g(M) \underline{\#}, \quad \underline{\#} \quad \vdash_U^*$$
$$h, \quad \# g(u) \underline{a} |^{i+1} g(v), \quad \# g(M) \underline{\#}, \quad \underline{\#}$$

and

iff M started on input $s_M, \#w\underline{\#}$ never halts.
iff U started on

$$s_U, \quad \# g(\#w\#) \underline{\#}, \quad \# g(M) \underline{\#}, \quad \underline{\#}$$

never halts.

Thus:

- If the head of M is located on a cell with a_i,
- then a_i is gödelised by $a|^{i+1}$, and
- the head of tape 1 of U is located on the first letter 'a' of $g(a_i)$.

Overview of $\mathcal{U}$

We use NOP for the DTM that does nothing:

$$NOP \;:=\; > \; R^{(1)}L^{(1)}$$

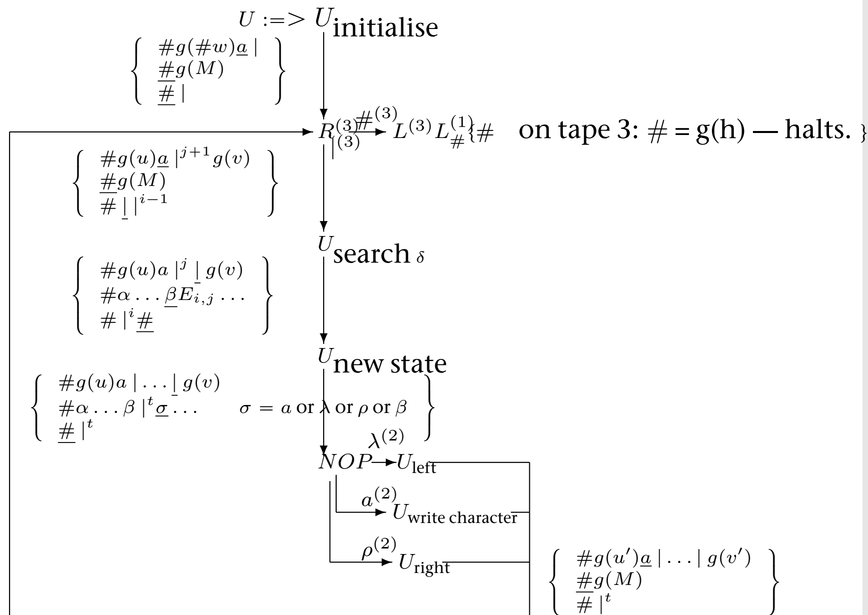

Figure 3: The structure of the universal DTM U

The submachine $U_{initialise}$:
When U starts, the 3 tapes are:

$$\#g(\#w\#)\underline{\#}$$
$$\#g(\text{M})\underline{\#}$$
$$\underline{\#}$$

$$U_{initialise} := \quad > L^{(2)}_{\#} \longrightarrow R^{(3)}|^{(3)} L^{(3)} \longrightarrow L^{(1)} L^{(1)}$$

Figure 4: The submachine $U_{initialise}$

$U_{initialise}$ works as follows:

- The head on tape 2 is immediately left to $g(\mathrm{M})$.
- It writes the number of the starting state on tape 3: $|$.
- On tape 1 the head is put on the gödelisation of the symbol, on which the head of M sits,
  namely on the first symbol of the gödelisation: ’a’.

So the tapes look as follows:

$$\#g(\#w)\underline{a}|$$
$$\underline{\#}g(\mathrm{M})\#$$
$$\underline{\#}|$$

Assume,

- M is in state q_i, and
- M sees the symbol a_j.

Then the tapes are as follows, when U_{search_δ} starts to work:

$$\begin{array}{r} \#g(u)\underline{a}\,|^{j+1}g(v) \\ \underline{\#}g(M) \\ \#\,\underline{|}\,|^{i-1} \end{array}$$

$U_{\text{search }\delta} :=$ $> R_\alpha^{(2)} R^{(3)}$ $|^{(3)}$ $\#^{(3)}$ {right line i} {search in $g(M)$}

$NOP \longrightarrow R^{(1)} \xrightarrow{|^{(1)}} R_\beta^{(2)}$ {right entry $E_{i,j}$} {to serach in line i}

$\neq |^{(1)}$ $L^{(1)}$

Figure 5: The submachine $U_{search\ \delta}$

The machine $U_{search\ \delta}$:

$U_{search\ \delta}$ does the following:

- It looks for the i-te line, which corresponds to q_i, by counting the α.
- Then it looks for the entry of q_i and a_j, by counting the occurrences of β.

The tapes of U now look:

$$\#g(u)a\ |^j \underline{|} g(v)$$
$$\alpha \ldots \underline{\beta} E_{i,j}$$
$$\#\ |^i \underline{\#}$$

The machine $U_{newstate}$:

When $U_{newstate}$ starts:

$$\#g(u)a\,|^{j}\underline{|}\,g(v)$$
$$\#\alpha\ldots\underline{\beta}E_{i,j}\ldots$$
$$\#\,|^{i}\underline{\#}$$

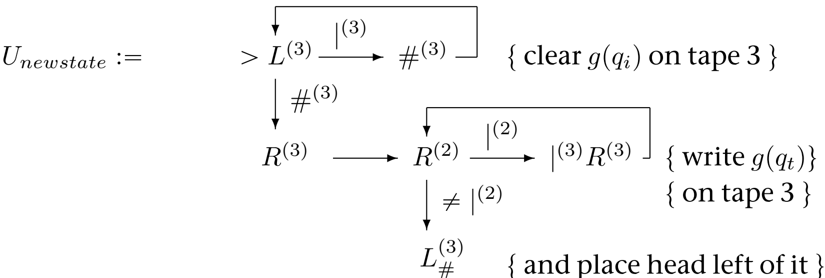

Figure 6: The machine $U_{newstate}$

$U_{newstate}$ does the following:

- The old state is removed from tape 3
- The entry $E_{i,j}$ starts with a number of strokes: the new state. It ends with the first non-stroke symbol.
  The new state is copied to tape 3.

The tapes of U look as follows:

$$\#g(u)a\,|^j\underline{|}\,g(v)$$

$$\#\alpha\ldots\beta\,|^t\underline{\Sigma}\ldots \qquad \Sigma = a \text{ or } \lambda \text{ oder } \rho \text{ or } \beta$$

$$\underline{\#}\,|^t$$

If $\Sigma = \beta$, then the entry $E_{i,j}$ was empty and U hangs, like the simulated machine M.

The machine U_{left}:

When U_{left} starts, the tapes are:

$$\#g(u)a\,|^j\underline{|}\,g(v)$$
$$\#\alpha\ldots\beta\,|^t\underline{\lambda}\ldots$$
$$\underline{\#}\,|^t$$

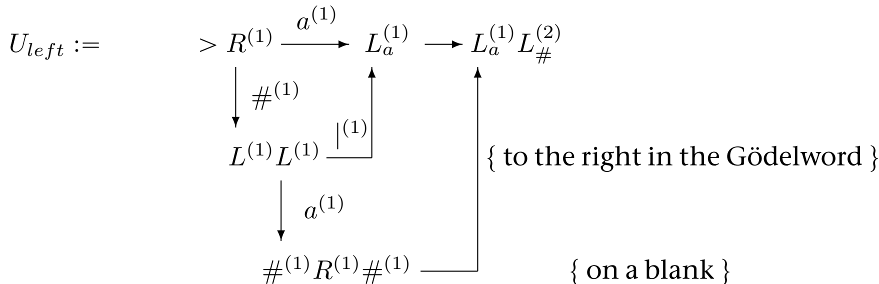

Figure 7: The machine U_{left}

U_{left} does the following:

- The gödelisation of each symbol starts with 'a'.
  To simulate a step of M to the left, U_{left} takes two steps to the left, to the next 'a'.
- Special case: U is on the blank of a gödelword to the extreme right of the described tape.
  Tape 1 in this case:

$$\#g(u)a\underline{|}\#\ldots$$

  U has to clear the last '$a|$'.
- U moves the head of tape 2 to the left immediately before $g()$.

The tapes of U are:

$$\#g(u')\underline{a}\,|\ldots|\,g(v')$$
$$\underline{\#}g(\mathrm{M})$$
$$\underline{\#}\,|^{t}$$

The machine U_{right}:

When U_{right} starts:

$$\#g(u)a\,|^{j}\underline{|}\;g(v)$$

$$\#\alpha\ldots\beta\,|^{t}\underline{\rho}\ldots$$

$$\underline{\#}\,|^{t}$$

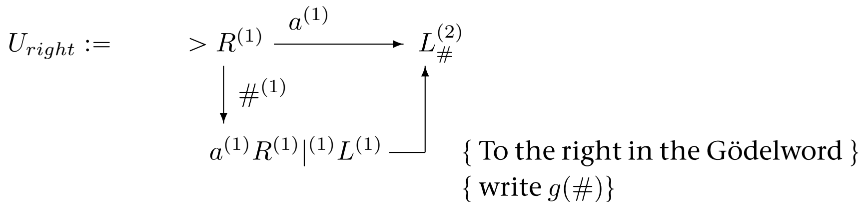

Figure 8: The machine U_{right}

U_{right} does the following:

- The head of U of tape 1 is on the last stroke of the actual symbol.
  The next symbol to the right is a 'a', the start of the next letter.
- If there is no 'a', then U is on a blank to the right of the gödelword of the tape contents. U writes $g(\#) = a|$.
- U moves the head of tape 2 to the left before $g(\mathrm{M})$.

The tapes of U are:

$$\#g(u')\underline{a}\,|\ldots|\,g(v')$$

$$\#g(\mathrm{M})$$

The machine $U_{writesymbol}$:
When $U_{writesymbol}$ starts:

$$\#g(u)a\,|^j\underline{|}\,g(v)$$
$$\#\alpha\ldots\beta\,|^t\underline{a}\ldots$$
$$\underline{\#}\,|^t$$

$U_{\text{writesymbol}} :=$

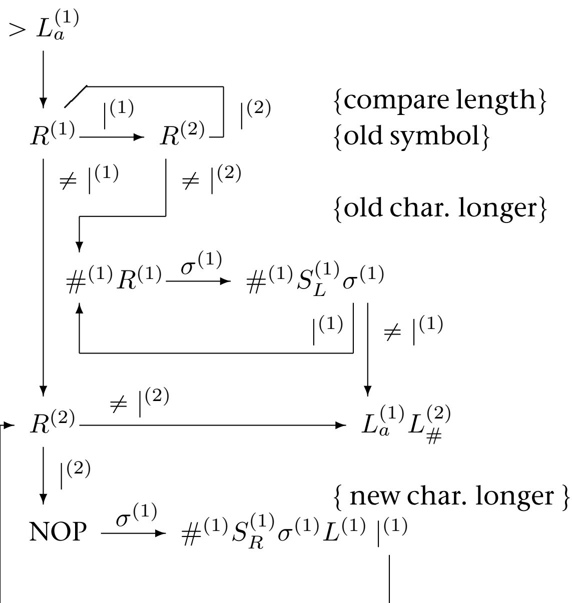

Figure 9: The machine $U_{writesymbol}$

$U_{writesymbol}$ does the following:

- The gödelword $a|^{j+1}$ on tape 1 has to be replaced by the gödelword $a|^{r+1}$.
- $j < r$: Gödelisation of the new letter is longer than the old one.

  $g(v)$ has to be shifted to the right.
  - For each "‘additional"’ stroke on tape 2:
  - Shift $g(v)$ by one cell to the right:
    Use the shift-right machine S_R.
    Separate the word-to-be-shifted with $\#$, remember the overwritten symbol.
  - Write a ’$|$’ in the free cell.

- $j > r$: Gödelisation of the new symbol is shorter than the one of the old one. $g(v)$ has to be shifted to the left.
  - Here only strokes have to be removed.
    $g(v)$ starts with a or $\#$ an.
  - Shift the word to the right of the head of tape 1 by one cell to the left.
    Use the shift-left machine S_L.
    Separate the word to be shifted with $\#$, remember the overwritten symbol.
  - Repeat shift-left, until a non-stroke is reached.
- Finally: Position the head of tape 1 to the a, with which the current tape symbol starts. Head of tape 2left before $g(M)$.

The tapes of U are:

$$\#g(u)\underline{a}\,|\ldots|\,g(v)$$
$$\underline{\#}g(\mathrm{M})$$
$$\underline{\#}\,|^{t}$$

Summary:

- We use for the simulated DTM M a gödelisation via a word over $\{\alpha, \beta, \rho, \lambda, a, 0, |\}$.
- α, β are separators between the descriptions of the δ-transitions.
  Then U can easily find the right δ-transition.
- The input word is gödelised with the same method as the DTM M. The alphabet is $\{a, 0, |\}$.

- In each step U works as follows:
  - Given the state of M and the currently scanned symbol of M, U searches for the gödelisation of the right δ-transition.
  - It changes the state according to the entry in $g(\mathrm{M})$.
  - It changes the contents of the tape according to the entry in $g(\mathrm{M})$.

    When the tape symbol is changed, then the gödelword of the tape contents right to the head has to be moved to the right or the left: Different tape symbols have gödelisations of different length.

2.2 PCP

Content of this section:

1. Posts Correspondence problem is an interesting example of an undecidable problem.
2. It can be reduced to many problems (which are therefore also undecidable).
3. In particular PCP can be reduced to the problem whether a contextfree grammar is unique or not.

Definition 2.5 (Correspondence system, PCP)

A correspondence system P over an alphabet Σ is a finite, indexed set of rules:

$$P = \{\langle u_1, v_1\rangle_1, \langle u_2, v_2\rangle_2, \ldots, \langle u_k, v_k\rangle_k\}$$

with $u_i, v_i \in \Sigma^*$.
A solution of P is a finite sequence of indices $i_1, \ldots, i_n$ $(1 \leqslant i_1, \ldots, i_n \leqslant k)$ such that

$$u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$$

This PCP problem (Post's Correspondence Problem) is the following. Given any correspondence system P over an alphabet Σ, determine whether there is a solution of it.

- Emil Post.
- A variant of a recursively unsolvable problem.
- Bulletin of the AMS, 52:264–268, 1946.

Definition 2.6 (Variants of PCP: 01-PCP, PCP$_k$)

The problem 01-*PCP* is the PCP problem over the alphabet $\{0, 1\}$.
The problem *PCP*$_k$, for fixed $k \in \mathbb{N}$, is a PCP where each input consists of exactly k pairs.

Example 2.7

What about the following correspondence systems?

$$
\begin{aligned}
P_1 &= \{\langle a, ab\rangle_1, \langle b, ca\rangle_2, \langle ca, a\rangle_3, \langle abc, c\rangle_4\} \\
P_2 &= \{\langle ab, a\rangle_1, \langle bcc, bbc\rangle_2, \langle ba, cb\rangle_3, \langle c, babc\rangle_4\}
\end{aligned}
$$

P_1: 12314, P_2: 123114

Example 2.8

Some more correspondence systems.

$$\begin{aligned} P_3 &= \{\langle a^2, a^2b\rangle_1, \langle b^2, ba\rangle_2, \langle ab^2, b\rangle_3, \langle abc, c\rangle_4\} \\ P_4 &= \{\langle a^2b, a^2\rangle_1, \langle a, ba^2\rangle_2\} \end{aligned}$$

P_3: 1213, P_4: no solution **Exercise**

Lemma 2.9 (PCP over one alphabet)

The PCP over a one element alphabet ($|\Sigma| = 1$) is decidable.

Proof: Exercise.

Note that when a PCP has one solution, then it has infinitely many. Can you find a solution of the following one:

$$(001, 0), (01, 011), (01, 101), (10, 001)$$

The shortest solution has length 66 and starts with 2,4,3,4,4,2,1,2,4,3.
This one is even worse

$$(0, 0000), (00, 001), (0111, 000), (1001, 1)$$

The shortest solution has length 698!!

Theorem 2.10 (Undecidability of PCP)

There is no algorithm to determine for a given correspondence system P whether it has a solution or not: PCP is undecidable.

Proof (Part 1).

We consider the following variant of PCP, called MPCP. Input is as for PCP, but we ask whether there is a solution $i_1, \ldots, i_n$ with $i_1 = 1$.

We show that

- H $\leq$ MPCP,
- MPCP $\leq$ PCP.

Proof (Part 2).

We introduce new symbols \$ and # and denote for $w = a_1 \dots a_m \in \Sigma^+$:

$$\begin{array}{lcl} \overline{w} & := & \#a_1\# \dots \#a_m\# \\ \acute{w} & := & a_1\# \dots \#a_m\# \\ \grave{w} & := & \#a_1\# \dots \#a_m \end{array}$$

To each instance $I = (x_1, y_1), \dots, (x_k, y_k)$ of length k for MPCP we assign an instance $f(I)$ of length $k+1$

$$f(I) := (\overline{x_1}, \grave{y}_1), (\acute{x}_1, \grave{y}_1), (\acute{x}_2, \grave{y}_2), \dots, (\acute{x}_k, \grave{y}_k), (\$, \#\$)$$

Proof (Part 3).

I has a solution with $i_1 = 1$ iff $f(I)$ has (any) solution. The idea is to use the fact that all LHS's end with the # while all RHS's start with it.

To start with a # we need to use the 0'th pair as a start and the $k+1$'th as the last (for the last #).

To be precise, (1) if I has a solution $1, i_1, \ldots, i_r$, then $f(I)$ has a solution $0, i_1, \ldots, i_r, k+1$ and (2) if $f(I)$ has a solution $i_1, \ldots, i_r$ then I has a solution $1, i_2 \ldots, i_{r-1}$.

Therefore MPCP $\leq$ PCP.

Proof (Part 4).

It remains to show H $\leq$ MPCP.

For each DTM M and input w we have to construct a correspondence system $(x_1, y_1), \ldots, (x_n, y_n)$ such that this system has a solution iff M halts on w.

The MPCP is over the alphabet $K \cup \Sigma \cup \{\#\}$: words code configurations.

Suppose M halts on w. Then there is a series of configurations from the initial one into an accepting one. We have to simulate this in our sequence of indices. □

Proof (Part 5).

The sequences we build should look as follows

$$\begin{aligned} x_{i_1} \ldots x_{i_j} &= \quad \#s_0\#s_1\#s_2\# \\ y_{i_1} \ldots y_{i_j} &= \#s_0\#s_1\#s_2\#s_3\# \end{aligned}$$

I.e. the first is one configuration short of the second. We start with $(\#, \#wq_0\#)$ (MPCP).

We need to find pairs (x, y) to simulate the running of the DTM and to construct

LHS: $\#wq_0\# \ldots \#\alpha_{k-1}q_{k-1}\beta_{k-1}\#$
RHS: $\#wq_0\# \ldots \#\alpha_k q_k \beta_k\#$

Proof (Part 6).

The following pairs allow us to achieve this:

- $(\#, \#)$, (x, x) for $x \in \Gamma$,
- for each $q \neq h$, p any state (including h), $x, y, z \in \Gamma$:

$$\begin{array}{ll} (qx, py) & \text{if } \delta(q, x) = (p, y) \\ (qx, xp) & \text{if } \delta(q, x) = (p, R) \\ (xq, px) & \text{if } \delta(q, x) = (p, L) \end{array}$$

By induction on k one verifies: If the DTM accepts w then we can build a sequence

$$\begin{array}{ll} \text{LHS:} & \#wq_0\#\ldots\#\alpha_{k-1}q_{k-1}\beta_{k-1}\# \\ \text{RHS:} & \#wq_0\#\ldots\#\alpha_k h \beta_k\# \end{array}$$

Proof (Part 7).

We need to make both sides identical in order to get a solution to the MPCP.

Up to now, all pairs ensure that the LHS can not get longer than the RHS. The following pairs make sure that this can be repaired:

$$(xhy, h),\ (hy, h),\ (xh, h),\ \text{and}\ (h\#\#, \#).$$

With these pairs, we can always extend the LHS such that the two sides get identical. In addition, we can easily state additional pairs in the MPCP such that this can be achieved.

$\leadsto$ **blackboard 2.1**

Lemma 2.11 (Undecidability of 01-PCP)

PCP can be reduced to 01-PCP, therefore 01-PCP is undecidable as well.

Proof: **Exercise**

Lemma 2.12 (Undecidability of PCP$_7$ (Senizergues, Matjiasevic, 1996))

The problem PCP$_k$, for fixed $k \geqslant 7$, is undecidable

Lemma 2.13 (Decidability of PCP$_2$ (Ehrenfeucht, Karhumaki, Rozenberg, 1982))

The problems PCP$_1$, PCP$_2$ are decidable.

Nearly proved in *Stacs 2015*, that also PCP$_5$ and PCP$_6$ are undecidable.

Theorem 2.14 (Undecidability of various problems)

Let G, G_1, G_2 be contextfree *grammars. Then the following problems are* undecidable*:*

1. *Is* $L(G_1) \cap L(G_2) = \emptyset$*?*
2. *Is* $|L(G_1) \cap L(G_2)|$ *infinite?*
3. *Is* $L(G_1) \cap L(G_2)$ *contextfree?*
4. *Is* $L(G_1) \subseteq L(G_2)$*?*
5. *Is* $L(G_1) = L(G_2)$*?*
6. *Is* $\overline{L(G)}$ *deterministic contextfree?*
7. *Is* $L(G)$ *regular?*
8. *Is* $L(G)$ *deterministic contextfree?*

Proof (Part 1).

We assign to each PCP $I = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ **over** $\{0, 1\}$ **two contextfree grammars** G_1, G_2 **over** $\{0, 1, \$, a_1, \ldots, a_k\}$ **as follows.** G_1 consists of the rules

$$
\begin{array}{rcc}
S & \rightarrow & A\$B \\
A & \rightarrow & a_1 A x_1 \mid \ldots \mid a_k A x_k \\
A & \rightarrow & a_1 x_1 \mid \ldots \mid a_k x_k \\
B & \rightarrow & y_1^R B a_1 \mid \ldots \mid y_k^R B a_k \\
B & \rightarrow & y_1^R a_1 \mid \ldots \mid y_k^R a_k
\end{array}
$$

Obviously, $L(G_1)$ is as follows

$$\{a_{i_n} \ldots a_{i_1} x_{i_1} \ldots x_{i_n} \$ y_{j_m}^R \ldots y_{j_1}^R a_{j_1} \ldots a_{j_m} | n, m \geqslant 1 \leqslant i_p, j_q \leqslant k\}.$$

Proof (Part 2).

G_2 consists of the rules

$$\begin{array}{lcc} S & \rightarrow & a_1 S a_1 \,|\, \ldots \,|\, a_k S a_k \,|\, T \\ T & \rightarrow & 0T0 \,|\, 1T1 \,|\, \$ \end{array}$$

Obviously, $L(G_2) = \{uv\$v^R u^R | u \in \{a_1, \ldots, a_k\}^*, v \in \{0,1\}^*\}$.

- The function $I \mapsto (G_1, G_2)$ is a reduction of PCP to $L(G_1) \cap L(G_2) = \emptyset$.
- It is also a reduction of PCP to $L(G_1) \cap L(G_2) = \infty$ (if the intersection is nonempty, then it must be infinite).

Proof (Part 3).

For (4) and (5), we note that **both grammars determine languages that are deterministic contextfree** (and DCFL is closed under complements). Thus we can effectively construct G'_1, G'_2 with $L(G'_i) = \overline{L(G_i)}$. Then

$$\begin{aligned} L(G_1) \cap l(G_2) = \varnothing \quad &\leftrightarrow \quad L(G_1) \subseteq L(G'_2) \\ &\leftrightarrow \quad L(G_1) \cup L(G'_2) = L(G'_2) \\ &\leftrightarrow \quad L(G_3) = L(G'_2) \end{aligned}$$

G_3 is not necessarily deterministic contextfree but can be effectively constructed.

Thus $I \mapsto (G_1, G'_2)$ is a reduction of PCP to $L(G_1) \subseteq L(G_2)$ and $I \mapsto (G_3, G'_2)$ is a reduction of PCP to $L(G_1) = L(G_2)$. □

Proof (Part 4).

We can certainly effectively construct G_4 with $L(G_4) = L(G'_1) \cup L(G'_2)$.

Then $I \mapsto (G_4)$ is a reduction of PCP to "$\overline{L(G)}$ is deterministic contextfree" (because I has a solution **iff** $L(G_1) \cap L(G_2) = \overline{L(G_4)}$ is not contextfree).

It is immediate that $I \mapsto (G_4)$ is also a reduction of PCP to "$L(G)$ is regular" as well as to "$L(G)$ is deterministic contextfree".

Because the **two grammars constructed in the proof** of the last theorem generate **deterministic cf languages**, we get:

Corollary 2.15 (Undecidability of various problems (2))

Let G_1, G_2 be two cf grammars that correspond to DCFL languages. Then the following problems are undecidable:

1. *Is* $L(G_1) \cap L(G_2) = \emptyset$*?*
2. *Is* $|L(G_1) \cap L(G_2)|$ *infinite?*
3. *Is* $L(G_1) \cap L(G_2)$ *deterministic contextfree?*
4. *Is* $L(G_1) \cup L(G_2)$ *deterministic contextfree?*
5. *Is* $L(G_1) \cap L(G_2)$ *contextfree?*
6. *Is* $L(G_1) \cup L(G_2)$ *contextfree?*
7. *Is* $L(G_1) \subseteq L(G_2)$*?*
8. *Is* $L(G_1)$ *regular?*

Plane tiling

Given: A finite set of tiles of different shape.
Wanted: Tiles can be put together, provided they fit together (valid tiling).

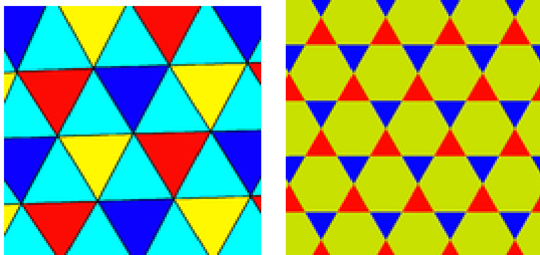

Figure 10: Simple periodic tilings.

2.3 Tiling the Plane

Aperiodic Plane tiling

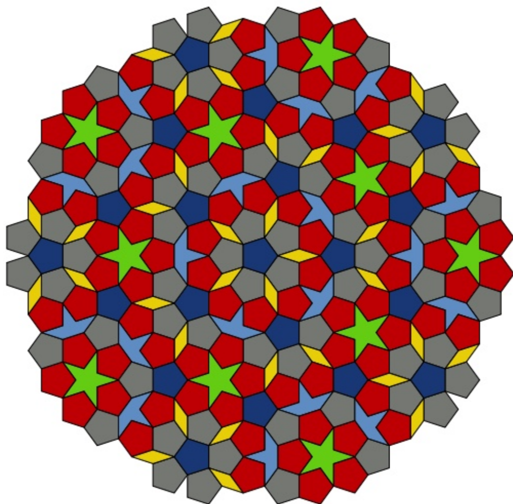

Figure 11: Aperiodic tiling (Penrose).

Aperiodic Plane tiling

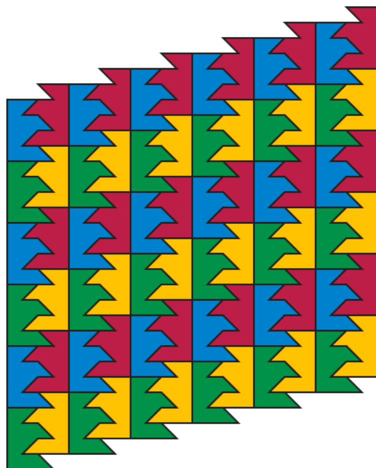

Figure 12: Tiling of the plane after Heesch.

Are there sets of tiles that allow **only aperiodic tilings**? Here is a set of tiles introduced by Robinson with this property.

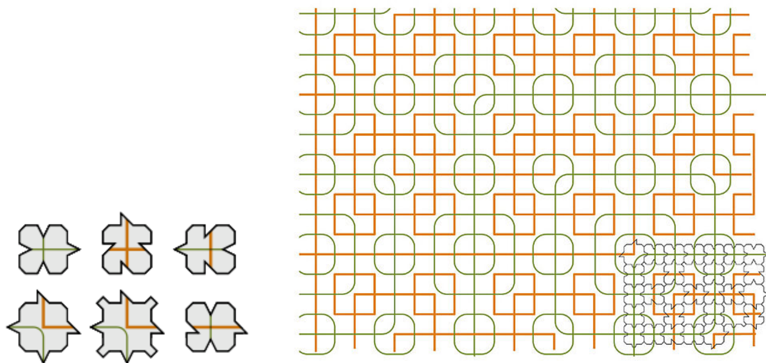

Figure 13: Robinson tiles and tiling.

The Domino-Problem (1)

- Plane tiling dates back to Hilbert's 18th problem (and was solved by Reinhardt in 1928).
- Hao Wang considered it in 1961 again and came up with an algorithm to compute a periodic tiling or give out no if it does not exist (for any set of tiles). However, he assumed that if the plane is tileable, it is also periodically tileable.
- In 1964, one of his PhD students showed the problem to be undecidable (and thus the assumption to be false).

The Domino-Problem (2)

Given: A finite set of different quadratic tiles, edges are colored (Wang tiles).

Wanted: Tiles can be put together, provided the colors of the edges match (valid tiling). Tiles can **not be rotated**.

The following set tiles the plane aperiodically.

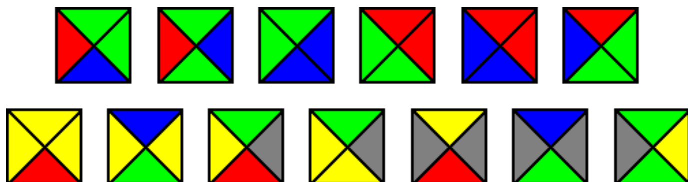

Figure 14: A set of Wang tiles.

The Domino-Problem (3)

Plane Tiling: Can the whole plane be covered with tiles (tiling of the plane)? We assume we have **infinitely many** tiles of each sort.

Finite Plane tiling: Given a finite set of Wang tiles and a color col. Is there a tiling of a finite rectangular part, where the boundary is colored with col?

Periodic: A tiling is periodic, if there exists $p, q \in \mathbb{N}$, such that the tiles at positions (i, j) and $(i + p, j + q)$ are identical (for all $i, j \in \mathbb{Z}$).

Aperiodic: A tiling is aperiodic if it is not periodic and there is no arbitrarily large part that is periodic.

Periodic Plane Tiling: Given a finite set of Wang tiles, is there a periodic tiling of the plane?

The Domino-Problem (3)

For fixed p, q, $\langle p, q \rangle$-tileability of the plane is decidable (try all finitely many possibilities of the $p \times q$ rectangle).

Lemma 2.16 (Periodic Plane Tiling)

Periodic plane tiling is recursively enumerable.

Lemma 2.17 (Plane Tiling)

If plane tiling is not possible, then there exists a finite rectangle of the plane that cannot be tiled.

Corollary 2.18

Plane tiling is co-recursively enumerable.

Proof of Corollary 2.18: $\rightsquigarrow$ **exercise**.

Proof of Lemma 2.17 (1).

Let a set of tiles be given.

We construct a theory T such that (1) **any model of T induces a plane tiling**, (2) **any unsatisfiable finite subset T_0 of T induces a finite rectangle that cannot be tiled**.

Then we apply the compactness theorem: T is satisfiable iff each finite subset is satisfiable. So if there is no plane tiling, T is not satisfiable, so there must be a finite subset T_0 of T that is not satisfiable. But then T_0 induces a finite rectangle that cannot be tiled.

Proof of Lemma 2.17 (2).

We represent a tile by the constant $t_{a,b,c,d}$ where a, b, c, d are from a finite set $Col = \{col_1, col_2, \ldots, col_{n_0}\}$: the first index represents *east*, the second *west*, the third *south* and the fourth *north* (i.e. the respective side of the tile). For the given set of tiles, we include the appropriate constants into T (as well as the negations of all possible tiles that are not given).

The fact that a tile is located at position i, j (where $i, j \in \mathbb{Z}$) is represented as $loc_{i,j}^{e,w,s,n}$. We add $loc_{i,j}^{e,w,s,n} \rightarrow t_{e,w,s,n}$ to T for all i, j, e, w, s, n (to make sure that such a tile exists): axioms for location.

Proof of Lemma 2.17 (3).

It remains to encode that we have a valid tiling: axioms for valid tiling. These axioms come in three groups.

We must ensure that there is exactly one tile on each position i, j: $loc_{i,j}^{col_1,col_1,col_1,col_1} \vee loc_{i,j}^{col_1,col_1,col_1,col_2} \vee \ldots \vee loc_{i,j}^{col_{n_0},col_{n_0},col_{n_0},col_{n_0}}$, and: $\neg(loc_{i,j}^{e,w,s,n} \wedge loc_{i,j}^{e',w',s',n'})$ for $\langle e, w, s, n\rangle \neq \langle e', w', s', n'\rangle$. We introduce the notation $loc_{i,j}^{*,*,*,*}$ for the first disjunction. If we fix one colour, we use $loc_{i,j}^{col_1,*,*,*}$ for the appropriate disjunction.

The tiles *fit* to each other:
$loc_{i,j}^{e,w,s,n} \rightarrow (loc_{i+1,j}^{*,e,*,*} \wedge loc_{i,j+1}^{*,*,n,*} \wedge loc_{i,j-1}^{*,*,*,s} \wedge loc_{i-1,j}^{w,*,*,*})$.

Obviously, by construction, any model of T induces a $\langle p, q\rangle$-tiling of the plane (by the $loc_{i,j}^{e,w,s,n}$ that are true).

Proof of Lemma 2.17 (4).

Suppose T_0 is a finite subset of T that is unsatisfiable. So T_0 contains finitely many axioms. **Let i_{min}, i_{max} (resp. j_{min}, j_{max}) the minimal and maximal ocurring i index (resp. j index).** We consider the rectangle with lowest left corner $\langle i_{min}, j_{min} \rangle$ and highest right corner $\langle i_{max}, j_{max} \rangle$.

If this rectangle were tileable, then T_0 would be satisfiable (in fact, an extension of T_0, namely the one that contains all $loc_{i,j}^{e,w,s,n}$ facts of the tiling, would be satisfiable). This is a contradiction.

□

Theorem 2.19 (Un-) Decidability of Plane Tiling)

Plane tiling as well as finite plane tiling are undecidable.

Instead of the original problem, we consider the following, slightly modified one.

Definition 2.20 (Plane Tiling with Restricted Origin)

Given a set of tiles and one distinguished tile t_0, is there a tiling of the plane that **uses t_0 at least once**?

Idea of Undecidability Proof

Find for each DTM a set of tiles that can simulate the DTM: DTM M accepts a word w **iff** the corresponding set tiles the plane.

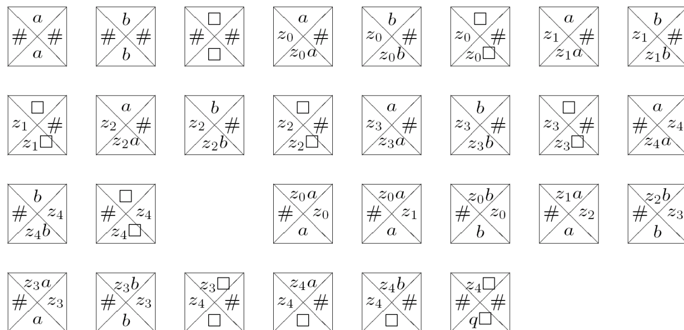

Figure 15: Simulating a DTM with Wang Tiles.

Lemma 2.21 (DTM described by Wang tiles)

For each DTM M **there is a finite set of tiles**, *including a distinguished tile* t_0, *such that the following are equivalent:*

- *the plane can be tiled using tile* t_0 *at least once,*
- M **does not terminate on empty input.**

Corollary 2.22 (Plane Tiling with Restricted Origin)

Plane Tiling with Restricted Origin is undecidable: *Given a finite set of Wang tiles, there is no algorithm to decide whether the plane can be tiled with them (with a distinguished tile to be used at least once).*

Proof of Lemma 2.21 (1).

The first step is to show the following: For each DTM M there is a finite set of tiles, including a distinguished tile t_0 s.t.

- the upper right quarter of the plane can be tiled using tile t_0 at least once, iff
- the TM M does not terminate on empty input.

In the second step we add just one new tile with which we can tile the remaining three quarters. □

Proof of Lemma 2.21 (2).

The idea is to start with the **initial configuration of the one-way infinite tape at the bottom (x-axis)**. The next line describes the configuration after the first move, the following line the second move etc. Only when the machine never terminates (on empty input), will this quarter of the plane be covered completely.

We need therefore

1. tiles for the initial configuration,
2. tiles to propagate the contents of the tape for "going up", and
3. tiles describing the moves of the DTM.

Proof of Lemma 2.21 (3).

Initial tiles: These are the only tiles with a “white” lower side: they must be placed on the x-axis. Only one tile has two “white” sides: it must be placed in the origin with t_0 next to it (exactly as the initial configuration of a DTM): (second one is t_0)

$$\overset{\#}{\square}\,*\,,\quad *\,\overset{(q_0,\#)}{\square}\,\diamond\,,\quad \diamond\,\overset{\#}{\square}\,\diamond$$

Tiles to go up: **“Copy the content of the tape to the next level”** (respect current position and current state).

$$\underset{a}{\overset{a}{\square}}\,,\quad p\,\underset{(p,a)}{\overset{a}{\square}}\,,\quad \underset{(p,a)}{\overset{a}{\square}}\,p$$

Proof of Lemma 2.21 (4).

Tiles describing δ: The first tile describes a simple **write action**, the second one a **move to the right**, the third one a **move to the left**. Note, that in our model we do not allow to **write and move in the same step**.

$$\underset{(q,a)}{\overset{(p,b)}{\square}} \quad \text{for } \delta(q,a) = (p,b),$$

$$\underset{(q,a)}{\overset{a}{\square}}\,{}_{p} \quad \text{for } \delta(q,a) = (p,R),$$

$${}_{p}\,\underset{(q,a)}{\overset{a}{\square}} \quad \text{for } \delta(q,a) = (p,L).$$

Proof of Lemma 2.21 (5).

To tile the upper quarter of the plane, the **lowest line is fully determined** (note we have to use t_0, which means it has to be the second tile on the bottom)

$$\overset{\#}{\square}{*} \quad {*}\overset{(q_0,\#)}{\square}{\diamond} \quad {\diamond}\overset{\#}{\square}{\diamond} \quad {\diamond}\overset{\#}{\square}{\diamond} \quad {\diamond}\overset{\#}{\square}{\diamond} \quad {\diamond}\overset{\#}{\square}{\diamond} \quad \ldots$$

Then the next line up is also determined, **when we require that exctly one tile in each line shows a pair (p, a) on its top**: above t_0 either one of the following types has to be placed:

$$\underset{(q_0,\#)}{\overset{(p,b)}{\square}}, \quad \underset{(q_0,\#)}{\overset{\#}{\square}}{}_p, \quad \text{or} \quad {}_p\underset{(q_0,\#)}{\overset{\#}{\square}}.$$

If it is of type 1, then all others are $\underset{\#}{\overset{\#}{\square}}$. If it is of type 2, then ${}_p\underset{a}{\overset{(p,a)}{\square}}$ has to be placed left to it, and all the others are again $\underset{\#}{\overset{\#}{\square}}$. For the third type, $\underset{(p,a)}{\overset{a}{\square}}{}_p$ must be placed as the first tile and all others are $\underset{\#}{\overset{\#}{\square}}$.

Proof of Lemma 2.21 (6).

$\rightsquigarrow$ **blackboard 2.2**

We also note that *there is a tiling (with t_0)* iff *there is a tiling (with t_0) and exactly one tile in each line shows a pair (p, a) on its top* (the problem is that instead of the tiles $\overset{a}{\underset{a}{\square}}$ one could also place alternately the tiles $\overset{(p,a)}{\underset{a}{\square}}{}_p$ and ${}_p\overset{(p,a)}{\underset{a}{\square}}$).

By induction, one shows easily that **each line corresponds exactly to a configuration** of the DTM started on empty input. Thus the whole quarter will be tiled iff the DTM does never terminate.

The last step is to tile the whole plane. This can be obtained by simply adding the tile $\square$, which is white at all sides. This allows to tile the remaining three quarters of the plane with only this tile. $\square$

Domino-Problem for finite areas (1)

Fixed rectangle: We fix a finite rectangular area of the plane ($n, m \in \mathbb{N}$) and a particular color col for the boundary. The Domino problem turns out to be NP-complete.

Quasi-fixed rectangle: We fix $n \in \mathbb{N}$ and two colors col_1, col_2. We also fix two n-vectors of colors. Is there an $m \in \mathbb{N}$ and a tiling of the $n \times m$ rectangle respecting the two vectors (as upper and lower parts) and the two colors on the two sides? This problem is PSPACE-complete.

Domino-Problem for finite areas (2)

Square: Suppose we fix an n_0-vector of colors and a particular color col. Is there $n \in \mathbb{N}$ and a tiling of the $n \times n$ square that respects the n_0-vector (as initial segment of the upper side and color col on all the remaining boundary? This problem is NEXPTIME-complete.

2.4 Recursive Functions

Content of this section:

1. We introduce the class of primitive recursive functions $\mathcal{PR}$ and the Grzegorczyk Hierarchy: $\bigcup_i \mathcal{E}^i = \mathcal{PR}$. The fourth class, $\mathcal{E}^3$ is also called the class of elementary functions.
2. The Ackermann function is not primitive recursive, but still DTM computable. This motivates the class of (partial) μ-recursive functions $\mathcal{R}$. This class corresponds to WHILE programs and functions computable by DTMs.
   $\mathcal{E}^{i+1}$ corresponds to LOOP programs with i nested LOOP statements.
3. Both primitive and μ-recursive functions satisfy certain boundedness conditions: they cannot grow as fast as possible. This is the reason why they are computable.

- In **Informatics 3** we introduced the important notion of a DTM computable function.
- Note that we also considered partial computable functions. They might not be always defined (when the DTM does not terminate).
- This and the next section is about different characterizations of this notion.
- We show that the μ-recursive functions are exactly the functions (1) computable by Turing machines, and (2) computable by WHILE programs.
- Note that a set W
  - is re <u>**iff**</u> there is a partial computable function f with $Def(f) = W$.
  - is recursive <u>**iff**</u> there is a total computable function f with $Range(f) = W$.

$\rightsquigarrow$ **blackboard 2.3**

Definition 2.23 (Primitive recursive functions $\mathcal{PR}$)

The class of primitive recursive functions is the smallest class that contains the functions listed below and is closed by applying the operations below.

1. Null 0: $\mathbb{N}^0 \to \mathbb{N},\ () \mapsto 0$,
2. Successor S: $\mathbb{N} \to \mathbb{N}, n \mapsto n+1$,
3. Projections π_i^k: $\mathbb{N}^k \to \mathbb{N}, \langle n_1, \ldots, n_k \rangle \mapsto n_i$ for $1 \leqslant i \leqslant k$.

Simultaneous substitution: If $g : \mathbb{N}^r \to \mathbb{N}$, $h_1 : \mathbb{N}^k \to \mathbb{N}$, $\ldots$, $h_r : \mathbb{N}^k \to \mathbb{N}$, are primitive recursive, so is

$$f : \mathbb{N}^k \to \mathbb{N};\ \overline{n} \mapsto g(h_1(\overline{n}), \ldots, h_r(\overline{n})),$$

Primitive Recursion: If $g : \mathbb{N}^k \to \mathbb{N}$, $h : \mathbb{N}^{k+2} \to \mathbb{N}$ are primitive recursive, so is $f : \mathbb{N}^{k+1} \to \mathbb{N}$ with

$$\begin{array}{lcl} f(\overline{n}, 0) & := & g(\overline{n}) \\ f(\overline{n}, m+1) & := & h(\overline{n}, m, f(\overline{n}, m)) \end{array}$$

What about the following definition of a function add? Is this a primitive recursive function?

$$\begin{array}{lcl} add(0,x) & = & x \\ add(y+1,x) & = & add(y,x)+1 \end{array}$$

Yes, but the definition above does not show it.

$$\begin{array}{lcl} add(x,0) & = & \pi_1^1(x) \\ add(x,y+1) & = & h(x,y,add(y,x)) \end{array}$$

where $h(x,y,z) = S(\pi_3^3(x,y,z))$.

Let $n \in \mathbb{N}$, $f : \mathbb{N}^n \to \mathbb{N}$, $h : \mathbb{N}^{n+2} \to \mathbb{N}$ and define the following function $g : \mathbb{N}^{n+1} \to \mathbb{N}$:

$$\begin{aligned} g(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ g(x_1, \ldots, x_n, y+1) &= h(y, x_1, \ldots, x_n, g(x_1, \ldots, x_n, y)) \end{aligned}$$

How to show that this function is primitive recursive?

Defining new functions using only Definition 2.23 can be difficult. With more functions and operations it would be easier.

Base functions: The functions $n+m$, -1, $n-m$, n^m, $n \times m$ etc are all in $\mathcal{PR}$.

Modifying variables: The scheme of primitive recursion is strict on the handling of arguments. For example $f(n) = n^2 + 1$ cannot be defined in that form (all functions have to be defined on all arguments).

Case distinction: A nice tool to have is definition by cases.

Sum and product: It would also be nice to allow building sums and products out of old functions.

Simultaneous recursion: The schema of primitive recursion only allows to define scalar functions. It would be better to have functions that map into $\mathbb{N}^k$.

These extensions are treated on the following slides.

Lemma 2.24 (More base functions)

The following functions are in $\mathcal{PR}$: $n+m$, $\dot{-}1$ (predecessor), $n \dot{-} m$ (modified subtraction), $|(n,m)|$, $n \times m$, n^m, $n!$.

Proof: $\leadsto$ **blackboard 2.4**

Lemma 2.25 (Modifying variables)

$\mathcal{PR}$ is closed under permutation, doubling and omitting of variables when using simultaneous substitution.

Proof: For a tuple $\overline{n}$ we can produce any $\overline{m_r}$ that is obtained by permuting or doubling certain variables by

$$\overline{m_r} = \langle \pi^k_{i_1}(\overline{n}), \ldots, \pi^k_{i_{t_r}}(\overline{n}) \rangle$$

Additional applications of the projections allow us also the omitting of variables.

Definition 2.26 (Case distinction)

Assume g_i, h_i, $1 \leqslant i \leqslant r$ are functions from $\mathbb{N}^k \to \mathbb{N}$ and for all $\overline{n}$ there is exactly one $1 \leqslant j \leqslant r$ with $h_j(\overline{n}) = 0$. Then we define the following function by case distinction:

$$f(n) = \begin{cases} g_1(\overline{n}), & \text{if } h_1(\overline{n}) = 0; \\ \vdots & \vdots \\ g_r(\overline{n}), & \text{if } h_r(\overline{n}) = 0. \end{cases}$$

Lemma 2.27 ($\mathcal{PR}$ is closed under case distinction)

Any class of functions which contains $\mathcal{PR}$ and is closed under substitution and prim. recursion is also closed under case distinction.

Proof.

$$f(\overline{n}) = g_1(\overline{n})(1 \dot{-} h_1(\overline{n})) + \ldots + g_r(\overline{n})(1 \dot{-} h_r(\overline{n}))$$

Definition 2.28 (Bounded sum and product)

Let $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$. Then we define functions f_1, f_2 from $\mathbb{N}^{k+1}$ into $\mathbb{N}$ as follows:

$$f_1(\overline{n}, m) = \begin{cases} 0, & \text{if } m = 0; \\ \sum_{i \lneqq m} g(\overline{n}, i), & \text{if } m \gneqq 0. \end{cases}$$

$$f_2(\overline{n}, m) = \begin{cases} 0, & \text{if } m = 0; \\ \prod_{i \lneqq m} g(\overline{n}, i), & \text{if } m \gneqq 0. \end{cases}$$

Lemma 2.29 ($\mathcal{PR}$ closed under bounded sum/product)

Any class of functions which contains $\mathcal{PR}$ and is closed under substitution and prim. recursion is also closed under bounded sums and bounded products.

Proof: $\rightsquigarrow$ **exercise**

Definition 2.30 (Simultaneous recursion)

Assume there are functions $\mathbf{g} : \mathbb{N}^n \to \mathbb{N}^m$, $\mathbf{h} : \mathbb{N}^{n+1+m} \to \mathbb{N}^m$. Then there is exactly one function $\mathbf{f} : \mathbb{N}^{n+1} \to \mathbb{N}^m$ defined by

$$\begin{aligned} \mathbf{f}(\overline{x}, 0) &= \mathbf{g}(\overline{x}) \\ \mathbf{f}(\overline{x}, y+1) &= \mathbf{h}(\overline{x}, y, \mathbf{f}(\overline{x}, y)) \end{aligned}$$

Definition 2.31 ($\mathcal{PR}$ for non-scalar functions)

A function $\mathbf{f} : \mathbb{N}^n \rightarrow \mathbb{N}^m$ is called **primitive recursive**, if all component functions f_i, where $\mathbf{f}(\overline{x}) = (f_1(\overline{x}), \ldots, f_m(\overline{x}))$, are primitive recursive.

Lemma 2.32 ($\mathcal{PR}$ and simultaneous recursion)

Any class of functions which contains at least $\mathcal{PR}$ is **closed under simultaneous recursion** *as defined in Definition 2.30.*

Fibonacci numbers

How to define Fibonacci numbers

- using the **basic functionality**, and
- using *simultaneous recursion*?

The idea is to store the last two numbers and thus to use a function into $\mathbb{N}^2$:

$$\left(\begin{array}{ccc} f(0) & = & (\mathbf{1},0) \\ f(1) & = & (\mathbf{1},\mathbf{1}) \\ f(2) & = & (2,\mathbf{1}) \end{array}\right) \left(\begin{array}{ccc} F(0) & = & (1,0) \\ F(y+1) & = & (+(F(y)),\pi_1^2(F(y)) \\ +(x_1,x_2) & = & add(\pi_1^2(x_1,x_2),\pi_2^2(x_1,x_2)) \end{array}\right)$$

How to prove Lemma 2.32?

The idea is to code a vector by a single number.
Compare with gödelisation (see Slide 223).

Lemma 2.33 (Coding Functions)

For each $2 \leqslant m \in \mathbb{N}$ *there exist functions* $C_m : \mathbb{N}^m \to \mathbb{N}$ *and* $D_m : \mathbb{N} \to \mathbb{N}^m$ *with:*

1. D_m, C_m *are* bijections between $\mathbb{N}$ and $\mathbb{N}^m$.
2. D_m, C_m *are* primitive recursive.
3. $D_m \circ C_m = id = C_m \circ D_m$.

Proof of Lemma 2.33

First we reduce the statement to the case $m = 2$:

$$\begin{aligned} C_{m+1}(x_1, \ldots, x_{m+1}) &= C_m(x_1, \ldots, x_{m-1}, C_2(x_m, x_{m+1})) \\ D_{m+1}(z) &= (id, \ldots, id, D_2)D_m(z) \end{aligned}$$

These two functions are obviously inverse to each other. They are both primitive recursive if C_2 and D_2 are.

Proof of Lemma 2.33 (2)

For the case $m = 2$ we consider the enumeration as shown in Figure 16, which is defined by $C_2(x, y) = \sigma(z) + y$ where we set $\sigma(z) = \frac{(x+y)(x+y+1)}{2}$. $z = x + y$ is the number of the diagonal containing (x, y). We have $\sigma(z + 1) = \sigma(z) + z + 1$, thus σ is primitive recursive and so is C_2.

For fixed z, $C_2(x, y)$ takes on the values $\sigma(z),\ \sigma(z) + 1, \ldots, \sigma(z) + z$. But the next value is already $\sigma(z) + z + 1$ which is $\sigma(z + 1)$ so we have a bijection.

Proof of Lemma 2.33 (3)

Similarly we show that D_2 is primitive recursive: writing $k = C_2(x, y)$ as a function of k, we get $z(k) = \mu_{t<k}\{\sigma(t) \dot{-} k = 0\}$.

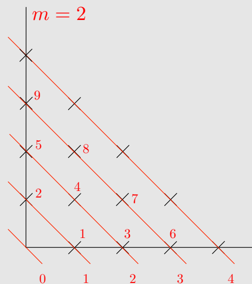

Figure 16: Bijection of $\mathbb{N}$ and $\mathbb{N}^2$.

Proof (of Lemma 2.32)

We define for $\overline{x} \in \mathbb{N}^n$:

$$g'(\overline{x}) = C_m\mathbf{g}(\overline{x}) \text{ and } h'(\overline{x}, y, z) = C_m\mathbf{h}(\overline{x}, y, D_m z)$$

Finally we define

$$\begin{aligned} f'(\overline{x}, 0) &= g'(\overline{x}) \\ f'(\overline{x}, y+1) &= h'(\overline{x}, y, f'(\overline{x}, y)) \end{aligned}$$

It remains to show that

$$\mathbf{f} = D_m \circ f'$$

(by induction on the variable y).

We consider the following functions Ack_n, called **branches of the Ackermann function.**

Definition 2.34 (Ackermann and its Branches)

For $n \in \mathbb{N}_0$ we define the following functions $Ack_n : \mathbb{N}^2 \to \mathbb{N}$:

$$
\begin{aligned}
Ack_0(x, y) &= y + 1 \\
Ack_{n+1}(x, 0) &= \begin{cases} x, & \text{if } n = 0; \\ 0, & \text{if } n = 1; \\ 1, & \text{else.} \end{cases} \\
Ack_{n+1}(x, y+1) &= Ack_n(x, Ack_{n+1}(x, y))
\end{aligned}
$$

The **Ackermann function** $Ack : \mathbb{N} \to \mathbb{N}$ is defined by $Ack(x) := Ack_x(x, x)$.

We get

- $Ack_0(x, y) = y + 1$,
- $Ack_1(x, y) = x + y$,
- $Ack_2(x, y) = xy$,
- $Ack_3(x, y) = x^y$.

Lemma 2.35 (Monotonicity of Ackermann)

*All **branches** Ack_n of the Ackermann function are **primitive recursive** functions. They are also **strictly monotone in both arguments**.*

$\rightsquigarrow$ **exercise**

- The function $Ack(n, n, n) : \mathbb{N} \to \mathbb{N}$ is not primitive recursive. It is contained in the larger class of recursive functions. **Proof later.**
- Grzegorczyk's idea: Use the Ack_n to build a hierarchy of classes which finally yield the whole class $\mathcal{PR}$: Definition 2.37.
- Define the classes analog to $\mathcal{PR}$ but with a limited form of primitive recursion plus the function Ack_n.
- Thus each subclass consists of prim. recursive functions that are dominated by a constant number of applications of Ack_n.
- The larger class $\mathcal{R}$ of (partial) recursive functions is obtained by adding another operation to $\mathcal{PR}$: the μ-operator.

$\rightsquigarrow$ **blackboard 2.5**

Definition 2.36 (Bounded operations)

The schema of primitive recursion in Definition 2.23 does not constrain the functions h, g. The following variant puts a bound on the function to be constructed.

Bounded Primitive Recursion: If $g : \mathbb{N}^k \to \mathbb{N}$, $h : \mathbb{N}^{k+2} \to \mathbb{N}$ and $b : \mathbb{N}^{k+1} \to \mathbb{N}$ are primitive recursive, so is $f : \mathbb{N}^{k+1} \to \mathbb{N}$ with

$$
\begin{array}{lcl}
f(\overline{n}, 0) & := & g(\overline{n}) \\
f(\overline{n}, m+1) & := & h(\overline{n}, m, f(\overline{n}, m)) \\
f(\overline{n}, m) & \leqslant & b(\overline{n}, m)
\end{array}
$$

Definition 2.37 (Grzegorczyk Hierarchy $\mathcal{E}^n$)

$\mathcal{E}^n$ is the smallest class of functions which contains the null function, the successor function, the projection functions π_i^k and the function Ack_n and is closed under substitution and bounded primitive recursion.

Theorem 2.38 (Growth)

$\mathcal{E}^n$: *for all* $f \in \mathcal{E}^n$ *there is* c_f *such that for all* $\overline{t}$*:*

$$f(\overline{t}) \leqslant Ack_{n+1}(\max\{2, \overline{t}\}, c_f)$$

Proof.

Induction on n and the construction of functions in $\mathcal{E}^n$.
Monotonicity properties of Ack_n are important too.
Note that for $f \in \mathcal{E}^0$, we can use the upper bound $\max\{1, \overline{t}\} + c_f$, for $f \in \mathcal{E}^1$ we use $(\max\{1, \overline{t}\}) \times c_f$ and for $f \in \mathcal{E}^2$ we use $(\max\{1, \overline{t}\})^{c_f}$. □

Corollary 2.39 (Grzegorczyk Hierarchy is strict)

$$\mathcal{E}^0 \subsetneqq \mathcal{E}^1 \subsetneqq \ldots \subsetneqq \mathcal{E}^n \subsetneqq \mathcal{E}^{n+1} \subsetneqq \mathcal{PR}$$

Proof.

Apply Theorem 2.38 and diagonalisation.

Assume $Ack_{n+1}(x,x) \in \mathcal{E}^n$. Then there is c_F such that $Ack_{n+1}(x,x) \leqslant Ack_{n+1}(x,c_F)$ for all $x \geqslant 2$. This is contradicting the monotonicity condition on the second argument. □

Example 2.40 (Fibonacci)

Consider the function $fib : \mathbb{N} \to \mathbb{N}$, which assigns to a number i the $i+1$'th Fibonacci number. Show that (1) $fib \in \mathcal{E}^3$, (2) $fib(2n) \geqslant 2^n$ for $n \geqslant 0$ and (3) $fib \notin \mathcal{E}^2$.

$\leadsto$ **exercise**

Here is another characterization of $\mathcal{E}^3$ which we note without proof.

Lemma 2.41 (Characterization of $\mathcal{E}^3 = \mathcal{E}$)

$\mathcal{E}^3$ is the smallest class of functions *containing the null function, the successor function, the projection functions, modified subtraction, and is closed under* substitution, bounded sum *and* bounded product.
Such functions are also called elementary.

$\rightsquigarrow$ **exercise**

1. Show that $f : \mathbb{N}^2 \rightarrow \mathbb{N}; (x, y) \mapsto x^{y+1}$ is in $\mathcal{E}$.
2. Show that $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by $f(0) = 1$, $f(n+1) = 2^{f(n)}$ is not in $\mathcal{E}$.

Non-elementary functions

Are non-elementary functions really needed in ordinary mathematics?

Theorem 2.42 (Grzegorczyk and primitive recursion)

The Grzegorczyk Hierarchy characterizes $\mathcal{PR}$, the class of primitive recursive functions:

$$\bigcup_{i=0}^{\infty} \mathcal{E}^i = \mathcal{PR}$$

Proof.

It suffices to show (**why?**) that for each prim. rec. function f there is a $n_f \in \mathbb{N}$ with

$$f(\bar{t}) \leqslant Ack_{n_f}(\max\{2, \bar{t}\}, n_f) \text{ for all } \bar{t} \in \mathbb{N}^k.$$

We show this by induction on the construction of prim. rec. functions.

Base functions: For S choose $n_S := 1$. This n_S works also for all other base functions.

Substitution: Let $f(\bar{t}) = g(f_1(\bar{t}), \ldots, f_k(\bar{t}))$. We choose $n_f := max(n_g, max(n_{f_1}, \ldots, n_{f_k}) - 1) + 2$. $\rightsquigarrow$ **blackboard 2.6**

Prim. Recursion: Let $f(\bar{t}, 0) = g(x)$, $f(\bar{t}, y + 1) = h(x, y, f(\bar{t}, y))$. We choose $n_f := max(n_g, n_h) + 3$. $\rightsquigarrow$ **exercise**

Often, the Ackermann-function is replaced by the following one, which is very similar.

Definition 2.43 (Ackermann-Péter function)

The *Ackermann-Péter* function $AP : \mathbb{N}^2 \to \mathbb{N}$ is defined as follows:

$$\begin{array}{lrcl} \text{AP-1} & AP(0,y) & = & y+1 \\ \text{AP-2} & AP(x+1,0) & = & AP(x,1) \\ \text{AP-3} & AP(x+1,y+1) & = & AP(x,AP(x+1,y)) \end{array}$$

$\rightsquigarrow$ **exercise**: Give a (good) lower bound on $AP(4,y)$.

Lemma 2.44 (Minimal set of initial functions for $\mathcal{PR}$)

To define the class $\mathcal{PR}$, not all projections

$$\pi_i^k$$

are needed. It suffices to have π_3^1 and π_{2i+5}^2, $i = 0, 1, \ldots$ as initial functions.
Together with null and successor, this set is minimal*:* *if* any of these functions is left out, *the* class obtained is strictly smaller *than the class of primitive recursive functions.*

- We defined the class $\mathcal{PR}$ and gave a hierarchy for it.
- **We are now switching to the next subsection and consider machines that are closer to real computers: random access machines.**
- We show that $\mathcal{PR}$ corresponds to LOOP programs. Indeed, $\mathcal{E}^n$ corresponds to LOOP programs with at most $n-1$ nested loops.
- But how to define the Ackermann function? We need WHILE constructs in our programs. Is there an equivalent extension of the class $\mathcal{PR}$?
- Yes, the class $\mathcal{R}$ of (partial) recursive functions. It is obtained from $\mathcal{PR}$ by adding the μ-operator. This operator in a sense simulates a WHILE loop.

We switch to Slides 343–351.

Theorem 2.45 ($\mathcal{PR}$ = LOOP)

The primitive recursive functions are exactly those that can be defined by LOOP programs.

Proof of Theorem 2.45: $\mathcal{PR} \subseteq$ LOOP.

Straightforward. Give **LOOP** programs for the base functions and show that the operations substitution and primitive recursion can be implemented by **LOOP** programs. Substitution of functions corresponds to composition of programs.

Proof of Theorem 2.45: LOOP $\subseteq \mathcal{PR}$ (1).

Induction on the construction of **LOOP** programs.

Problem: We have to deal with the registers of RAM's, but primitive recursive functions map into $\mathbb{N}$.

Solution: Use the coding functions of Definition 2.33 to code all registers into one natural number. Incrementing or decrementing these registers can be done with primitive recursive functions. Thus the **LOOP** constructs can be simulated by functions in $\mathcal{PR}$.

We finally have to show

$$P \mid_{[a_0,\ldots,a_k]} = [b_0,\ldots,b_k] \ \underline{\textbf{iff}}\ g_P(a_0,\ldots,a_k) = (b_0,\ldots,b_k)$$

Proof of Theorem 2.45: LOOP $\subseteq \mathcal{PR}$ (2).

The primitive recursion corresponding to a LOOP program P : $\mathbf{loop}\ x_i\ \mathbf{do}\ Q\ \mathbf{end}$ is as follows.

$$\begin{array}{rcl} h(x,0) & = & x \\ h(x,n+1) & = & g_Q(h(x,n)) \\ g_P(n) & = & h(n_i,n) \end{array}$$

Definition 2.46 (μ-Operator)

Let $g : \mathbb{N}^{k+1} \to \mathbb{N}$ be a partial function. We say that the function $f : \mathbb{N}^k \to \mathbb{N}$ is obtained from g by applying the unrestricted μ-operator if the following holds:

$$f(\overline{n}) = \mu_i\{g(\overline{n}, i) = 0\}$$

where μ is defined by

$$:= \begin{cases} i_0, & \text{if } g(\overline{n}, i_0) = 0 \text{ and} \\ & \forall\, 0 \leqslant j \lneqq i_0\text{: } g(\overline{n}, j) \text{ is defined and} \neq 0; \\ \text{undef}, & \text{otherwise.} \end{cases}$$

f is obtained from g by applying the normal μ-operator, if g is total, $f(\overline{n}) = \mu_i\{g(\overline{n}, i) = 0\}$ and for all $\overline{n}$ there is a $j \in \mathbb{N}$ with $g(\overline{n}, j) = 0$.

Definition 2.47 (μ-recursive functions, $\mathcal{R}, \mathcal{R}^{part}$)

The class of **total μ-recursive functions**, $\mathcal{R}$, is the smallest class which contains the primitive recursive functions and is closed under simultaneous substitution, primitive recursion and applying the **normal μ-operator**.

The class of **partial μ-recursive functions**, $\mathcal{R}^{part}$, is obtained by using the **unrestricted μ-operator** in the definition above.

- Why are we so picky about distinguishing both variants?
- Diagonalisation only leads to a contradiction if a function is total!
- M **prints** $f(n)$ **iff** M **prints** $f(n)+1$ is no contradiction, if M **does not halt at all**.
- The **unrestricted μ-operator** corresponds to arbitrary **WHILE** constructs.
- The **normal μ-operator** corresponds to **WHILE** constructs that always terminate

Back to Definition 2.56 on slide 352.

Theorem 2.48 ($\mathcal{R}$ = WHILE, $\mathcal{R}^{part}$ = WHILE^{part})

1. *The* total recursive functions *are exactly those total functions, that can be defined by* WHILE programs*.*
2. *The* partial recursive functions *are exactly those partial functions, that can be defined by* WHILE programs*.*

Proof of Theorem 2.48: $\mathcal{R} \subseteq$ WHILE.

The proof is an extension of the proof of Theorem 2.45.

The **WHILE** program corresponding to $g = \mu f$ is

$x_0 := 0$
$y := f(\overline{x}, x_0)$
while $y \neq 0$ **do**
  $x_0 := x_0 + 1$; $y := f(\overline{x}, x_0)$
end while

Proof of Theorem 2.48: WHILE $\subseteq \mathcal{R}$.

The μ-operator corresponding to a WHILE construct is done as
$\rightsquigarrow$ **exercise.**

Lemma 2.49 (Minimal set of initial functions for $\mathcal{R}^{part}$)

The class of **partial recursive functions** $\mathcal{R}^{part}$ *can also be defined in the spirit of Lemma 2.44.*

$\mathcal{R}^{part}$ *is the smallest class containing the null function, the successor function, the projection* π_3^1 *and any infinite subset of* π_i^2, $i \geqslant 2$ *as initial functions and which is closed under substitution, the unrestricted* μ*-operator and primitive recursion.*

This set is **minimal***: if any of these functions is left out, the class obtained is* **strictly smaller** *than* $\mathcal{R}^{part}$*.*

Theorem 2.50 (Grzegorczyk and LOOP programs)

The class $\mathcal{E}^{n+1}$ corresponds exactly to the class of all ***LOOP*** *programs with up to n nested LOOP statements.*

Proof.

By induction on n. The main part is to show that computing Ack_{n+1} given a **Loop** program for Ack_n can be done by wrapping the **Loop** program with another **Loop** construct. This increments the nesting by 1. $\leadsto$ **exercise** □

What does all this have to do with Turing machines?

Theorem 2.51 ($\mathcal{R}$ = DTM, $\mathcal{R}^{part}$ = DTM^{part})

1. *The total recursive functions are exactly those total functions, that can be defined by Turing machines.*
2. *The partial recursive functions are exactly those partial functions, that can be defined by Turing machines.*

Therefore all notions of computability are equivalent!

Proof.

(1): Using our previous results, it suffices to show that **GOTO** programs can be simulated by DTM's.
$\rightsquigarrow$ **exercise**

(2): Quite some coding!!

This finishes Sections 2.5 and 2.6.

Universal functions

- We know that there exists a universal DTM for the class $\mathcal{R}^{part}$. In other words, there is a partial recursive function which is universal for the class $\mathcal{R}^{part}$.
- Is there a universal function for the class $\mathcal{R}$ in $\mathcal{R}$?
- Is there a universal function for the class $\mathcal{PR}$ in $\mathcal{PR}$?

2.5 Random Access Machines

Content of this section:

1. We introduce Random Access Machines. These are more similar to real computers than Turing machines.
2. We also introduce three different classes of programs. They can be seen as simplified versions of Pascal or C programs.
3. LOOP programs correspond to primitive recursive functions.
4. WHILE programs correspond to general recursive functions.
5. GOTO programs are equivalent to **WHILE** programs.

Definition 2.52 (Random Access Machine)

A *random access machine (RAM)* consists of

1. a finite set of registers $x_1, x_2, \ldots, x_n$, where each register can store an arbitrarily large natural number,
2. a finite program (to be specified below).

Initially all registers are set to 0 (if not explicitly said otherwise). The machine follows program instructions in the order given.

We now describe several program constructs.

Definition 2.53 (LOOP construct)

LOOP programs are inductively defined:

Induction base: The following are LOOP statements as well as **LOOP** programs (for all x_i):

- $x_i := x_i + 1$,
- $x_i := x_i - 1$.

Induction step: If P_1 and P_2 are LOOP programs then

- $P_1; P_2$ is a **LOOP** program,
- loop x_i do P_1 end is a **LOOP** statement as well as a **LOOP** program for all registers x_i.

Definition 2.54 (Semantics of LOOP programs)

Let M be a RAM. M *executes LOOP programs* as follows:

- $x_i := x_i + 1$: M increments register x_i.
- $x_i := x_i - 1$: If $x_i > 0$ then M decrements register x_i. Otherwise x_i keeps the value 0.
- **loop** x_i **do** P_1 **end**: M executes P n-times, where n is the value stored in x_i **before** the first execution of the loop.
- $P_1; P_2$: M first executes P_1 and then, with the numbers stored then in the registers, the program P_2.
- Stop: If there are no more statements, the execution of the program terminates.

Definition 2.55 (LOOP computable)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is *LOOP computable*, if there is a RAM M with **LOOP** program P, such that for all $(n_1, \ldots, n_k) \in \mathbb{N}^k$ and all $m \in \mathbb{N}$:

$f(n_1, \ldots, n_k) = m \iff$
M started with input n_i in register x_i for $1 \leqslant i \leqslant k$ and 0 in all other registers, terminates with n_i in x_i for $1 \leqslant i \leqslant k$, m in register x_{k+1} and 0 in all remaining registers.

We denote by *LOOP* the set of all **LOOP** computable functions.

new command	simulation as LOOP command
$x_i := c$	//set x_i to zero $loop\ x_i\ do\ x_i := x_i - 1\ end;$ $\left.\begin{array}{c} x_i := x_i + 1; \\ \vdots \\ x_i := x_i + 1 \end{array}\right\} c -$ times
$x_i := x_j \pm c$	//set x_n to x_j $loop\ x_j\ do\ x_n := x_n + 1\ end;$ $x_i := 0;$ $loop\ x_n\ do\ x_i := x_i + 1\ end;$ $x_n := c;$ $loop\ x_n\ do\ x_i := x_i \pm 1\ end;$ $x_n := 0$

new command	simulation as LOOP command

$if\ x_i = 0\ then\ P_1\ else\ P_2\ end$	//If $x_i = 0$ then $x_n = 1$
	//and $x_{n+1} = 0$, and vice versa
	$x_n := 1;\ x_{n+1} := 1;$
	$loop\ x_i\ do\ x_n := x_n - 1\ end;$
	$loop\ x_n\ do\ x_{n+1} := x_{n+1} - 1\ end;$
	//execute P_1 respectively P_2
	$loop\ x_n\ do\ P_1\ end;$
	$loop\ x_{n+1}\ do\ P_2\ end;$
	$x_n := 0;\ x_{n+1} := 0$

$if\ x_i > c\ then\ P\ end$	$x_n := x_i - c;$ $loop\ x_n\ do\ x_{n+1} := 1\ end;$ $loop\ x_{n+1}\ do\ P\ end;$ $x_n := 0;\ x_{n+1} := 0$

new command	simulation as LOOP command
$x_i := x_j \pm x_k$	$x_i := x_j + 0;$ $loop\ x_k\ do\ x_i := x_i \pm 1\ end$
$x_i := x_j \cdot x_k$	$x_i := 0;$ $loop\ x_k\ do\ x_i := x_i + x_j\ end$
NOP	//This command does nothing $x_n := x_n - 1$

new command	simulation as LOOP command

$x_i := x_j\ DIV\ x_k$
$x_{i'} := x_j\ MOD\ x_k$
	//each time x_n counts up to x_k
	//If x_k is reached, x_j is once
	//more dividable by x_k
	//x_{n+1} is used to test whether $x_n = x_k$
	$x_i := 0;$
	$loop\ x_j\ do$
	$x_n := x_n + 1;$
	$x_{n+1} := x_k - x_n;$
	$if\ x_{n+1} = 0\ then\ x_i := x_i + 1;\ x_n := 0$
	$else\ NOP;$
	$endif$
	$enddo;$
	$x_{i'} := x_n;$
	$x_n := 0;\ x_{n+1} := 0$

And now back to Slide 327.

We extend the class of **LOOP** programs by introducing more constructs.

Definition 2.56 (WHILE construct)

WHILE programs are inductively defined

Induction base: The following are **WHILE** statements as well as **WHILE** programs (for all x_i):

- $x_i := x_i + 1$,
- $x_i := x_i - 1$.

Induction step: If P_1 and P_2 are **WHILE** programs then

- $P_1; P_2$ is a **WHILE** program,
- $\mathbf{while}\ x_i \neq 0\ \mathbf{do}\ P_1\ \mathbf{end}$ is a **WHILE** statement as well as a **WHILE** program for all registers x_i.

Question:

What is the semantics of **WHILE** programs?

Definition 2.57 (Semantics of the WHILE statement)

Let M be a RAM. M executes "**while** $x_i \neq 0$ **do** P_1 **end**" as follows:

1. If the value of x_i is different from 0, then M executes P, otherwise it jumps to step 3.
2. M repeats step 1.
3. M executes the statement right after the **WHILE** statement.

Definition 2.58 (WHILE computable)

A function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is called WHILE computable, if there is a RAM M and a **WHILE** program P, such that for all $(n_1, \ldots, n_k) \in \mathbb{N}^k$ and all $m \in \mathbb{N}$:

$f(n_1, \ldots, n_k) = m \iff$
If M is activated with n_i in register x_i for $1 \leqslant i \leqslant k$ and 0 in all other registers, then M terminates with n_i in x_i for $1 \leqslant i \leqslant k$, m in register x_{k+1} and 0 in all other registers.

$f(n_1, \ldots, n_k)$ undef $\iff$
M activated with n_i in register x_i for $1 \leqslant i \leqslant k$ and 0 in all other registers, does never terminate.

WHILE is the set of all total **WHILE** computable functions.
WHILE^{part} is the set of all partial **WHILE** computable functions.

- Can we implement the Ackermann function with a WHILE program?
- Can we implement each Ack_n with a LOOP program?

1. We implement Ackermann with stacks.
2. Stacks are not available in **WHILE** programs.
3. But we can simulate stacks as numbers using the coding functions and the fact, that all necessary operations on stacks can be done with LOOP programs.

Implementing Ackermann/Peter with stacks

For convenience we consider AP from Definition 2.43:

$$
\begin{array}{lllr l}
AP (& 2, & & 1\ ) & 2,1 \\
AP (& 1, & & AP(2,0)\ ) & 1,2,0 \\
AP (& 1, & & AP(1,1)\ ) & 1,1,1 \\
AP (& 1, & & AP(0,AP(1,0))\ ) & 1,0,1,0
\end{array}
$$

The stacks get larger and shorter and are not bounded.

An algorithm for computing AP with a stack S is shown on the next slide.

Implementing Ackermann/Peter with stacks

Let $S[]$ denote a stack. Initially its length is 2. When the algorithm terminates ($l = 1$), the value is stored in $S[1]$.

while $l > 1$ **do**
  if $S[l-1] = 0$ **then**
    $S[l-1] = S[l] + 1; l = l - 1$
  else if $S[l] = 0$ **then**
    $S[l] = 1; S[l-1] = S[l-1] - 1$
  else
    $S[l+1] = S[l] - 1; S[l] = S[l] - 1$
    $S[l-1] = S[l-1] - 1; l = l + 1$
  end if
end while

The idea is to use the coding functions from Lemma 2.33 to code tuples $\langle l_1, \ldots, l_m \rangle$ with C_m and to show, that *push* and *pop* on C_m can be realized as **Loop** programs.

A Beaver function

Length: Define the length $l(P)$ of **Loop** programs by counting the assignments and loops.

Normalized: Define normalized **Loop** programs by requiring that all used registers in P are among $x_1, \ldots, x_{2l(P)}$. There are only finitely many normalized programs of fixed length l.

Beaver: Let $val(P)$ by the value of x_1 after termination of P when started on empty registers. Define

$$\mathbf{B}(n) := \max_{\textbf{Loop}\text{ program } P} \{val(P) : P \text{ normalized of length} \leqslant n\}.$$

$\mathbf{B}$ is not primitive recursive, but while computable.

Definition 2.59 (GOTO construct)

GOTO programs are constructed as follows:

Index: An index j is any natural number: $j \in \mathbb{N}$.

GOTO statement: For a register x_i and an index j the following are **GOTO** statements:

- $x_i := x_i + 1$,
- $x_i := x_i - 1$,
- $\textbf{if } x_i = 0 \textbf{ goto } j$.

GOTO program: **GOTO** programs are defined as follows:

- $j : \ I$ is a **GOTO** program, if I is a **GOTO** statement,
- $P_1; P_2$ is a **GOTO** program, if P_1, P_2 are already **GOTO** programs.

Semantics of **GOTO** programs: as usual.

Theorem 2.60 (LOOP vs WHILE programs)

LOOP *programs can be simulated by* ***WHILE*** *programs.*
LOOP *computable functions are a* **strict subset** *of the set of* ***WHILE*** *programs.*

Proof.

It is easy to show that the **LOOP construct** can be simulated with three **WHILE constructs**.
We have seen that the Ackermann function is **WHILE** computable but not **LOOP** computable. □

Theorem 2.61 (WHILE=GOTO)

Given $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ the following are equivalent:

- *f is **WHILE** computable,*
- *f is **GOTO** computable,*

Corollary 2.62 (WHILE$^{\text{total}}$=GOTO$^{\text{total}}$)

The last theorem is also true when we consider all total *functions that can be built with the programming constructs.*

Proof of Theorem 2.61: (1) $\Rightarrow$ (2).

We have to simulate the **WHILE** construct with a **GOTO** program. Wlog we simulate **while** $x_i \neq 0$ **do** P **end** where P does not contain any **while**, x_{new} is a new register, j_1, j_2, j_3 are new indices. Let $\hat{P}$ be the program P with indices in front (starting with $1, 2, 3, \ldots$ until the last line). The **GOTO** program simulating the **WHILE** is

j_1 : **if** $x_i = 0$ **goto** j_3

$\hat{P}$

j_2 : **if** $x_{new} = 0$ **goto** j_1
j_3 : $x_{new} := x_{new} - 1$

Proof of Theorem 2.61: (2) ⇒ (1).

We cannot simply simulate each **GOTO** statement independently. We choose a new variable x_{count} s.t: **if its value is i then in the GOTO program to be simulated, the next line would be a GOTO statement to index i.**

For $P_i : \; x_i := x_i \pm 1$, let P'_i be $x_i := x_i \pm 1$;$x_{\text{count}} := x_{\text{count}} + 1$. If P_i is of the form if $x_i = 0$ goto j, let P'_i be

$$\text{if } x_i = 0 \text{ then } x_{\text{count}} := j \,; \text{ else } x_{\text{count}} := x_{\text{count}} + 1 \text{ end.}$$

The **WHILE** program looks then as follows:

$x_{\text{count}} := 1$
while $x_{\text{count}} \neq 0$ do
  if $x_{\text{count}} = 1$ then P'_1 end
  ⋮
  if $x_{\text{count}} = t$ then P'_t end
  if $x_{\text{count}} > 1$ then $x_{\text{count}} := 0$ end
end

We consider the **if** construct defined as a macro on Slide 351 with a LOOP program.

- It can also be defined with **WHILE** construct.
- However, it is more primitive (**WHILE** is not definable with **if**).
- We therefore define the class of $\mathbf{WHILE}^{\mathbf{if}}$ programs, where **if** is a new primitive construct.

Definition 2.63 (WHILE$^{\text{if}}$ programs)

WHILE$^{\text{if}}$ programs are inductively defined

Induction base: The following are **WHILE$^{\text{if}}$** statements as well as **WHILE$^{\text{if}}$** programs (for all x_i):

- $x_i := x_i + 1$,
- $x_i := x_i - 1$.

Induction step: If P_1 and P_2 are **WHILE$^{\text{if}}$** programs then

- "$P_1; P_2$" is a **WHILE$^{\text{if}}$** program,
- "**if** $x_i = n$ **then** P_1 **end**" is a **WHILE$^{\text{if}}$** program,
- **while** $x_i \neq 0$ **do** P_1 **end** is a **WHILE$^{\text{if}}$** statement as well as a **WHILE$^{\text{if}}$** program for all registers x_i.

Lemma 2.64 (WHILE=WHILE^{if})

__WHILE__ programs and __WHILE__^{if} programs are equivalent: they compute the same functions.

Theorem 2.65 (One WHILE loop suffices)

Each __WHILE__ computable function can be computed by a __WHILE__^{if} program with only one WHILE loop.

Proof.

Consider again Theorem 2.61. The theorem states that each **WHILE** program can be simulated by a **GOTO** program. The proof of Theorem 2.61 ((2) $\rightarrow$ (1)) shows that we can simulate this **GOTO** program by a **WHILE** program with just one **WHILE** loop. □

- Relation between WHILE computable functions and DTM-definable functions?
- We could show that **GOTO** programs can be simulated by Turing machines.
- The converse, however, is quite difficult.
- Instead, it is easier to show that

  1. $\mathcal{R} \subseteq$ **WHILE**, and
  2. $\mathcal{R}^{part} \subseteq$ **WHILE**, and then
  3. each total DTM computable function is in $\mathcal{R}$, and
  4. each partial DTM computable function is in $\mathcal{R}^{part}$.

Now back to Theorem 2.48 on Slide 334.

2.6 Hilberts 10. Problem

Content of this section:

1. We introduce Hilbert's 10. Problem.
2. Instead of solving a set of diophantine equations over $\mathbb{N}$, we can also solve a single diophantine equation over $\mathbb{Z}$.
3. We reduce the halting problem of RAMs to the solvability of a diophantine equation with exponentiation over $\mathbb{N}$.
4. Exponentiation can be eliminated (Matiyasevich).
5. Thus Hilbert's 10. Problem is unsolvable.

Hilbert (1900), ICM talk in Paris

10. Entscheidung der Lösbarkeit einer diophantischen Gleichung.

Eine diophantische Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoefficienten sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittels einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist.

Definition 2.66 (Diophantine Equation)

A diophantine equation is an equation of the form $p(x_1, \ldots, x_k) = 0$ where p is a polynomial in $x_1, \ldots, x_k$ with coefficients in $\mathbb{Z}$. Such an equation is solvable over $\mathbb{Z}$ (resp. $\mathbb{N}$) if there is a solution in $\mathbb{Z}^k$ (resp. $\mathbb{N}^k$).

$\rightsquigarrow$ **blackboard 2.7**

Definition 2.67 (Diophantine Set)

A set $M \subseteq \mathbb{N}$ is called diophantine if there is a diophantine equation $p(x_0, x_1, \ldots, x_k) = 0$ over $\mathbb{Z}$ with $M = \{x : \exists x_1 \ldots x_k \in \mathbb{Z} : p(x, x_1, \ldots, x_k) = 0\}$.

diophantine=re?

The following problems can be restated as the *unsolvability of a certain diophantine equation*:

- Goldbach conjecture,
- Riemann hypothesis,
- Four Color Conjecture.

In fact, *each Π_1^0 statement* (see Chapter 5) is equivalent to the *unsolvability of a particular diophantine equation*.

Lemma 2.68 (Reduction)

Solving a set of diophantine equations over $\mathbb{N}$ *can be reduced to solving a* single diophantine equation over $\mathbb{Z}$.

Solving sets of equations over $\mathbb{N}$ *or* $\mathbb{Z}$ *can be reduced to each other.*

$\rightsquigarrow$ **exercise**

Lemma 2.69 (Reduction to diophantine eq. with exp)

The halting problem for Random Access Machines can be reduced to solving diophantine equations with exponentiation over $\mathbb{N}$.

Proof.

Given any **GOTO** program P with m instructions over $n \leqslant m$ registers, we construct a set of diophantine equations with exp, such:

P terminates when started with empty registers

iff

the set of equations has a solution over $\mathbb{N}$

□

Proof of Lemma 2.69 (2).

Idea of the Reduction (1)

1. The variables in the equations to be constructed contain the values of all registers for each timepoint of the computation.
2. We encode such values wrt to a suitable basis B (large enough):

$$\sum_{i=0}^{S} a_i B^i,$$

S: number of steps until termination.

Proof of Lemma 2.69 (3).

Idea of the Reduction (2)

1. We use the notion $n \trianglelefteq n'$ to denote dominance: if $n = \sum_{i=0}^{S} a_i B^i$ and $n' = \sum_{i=0}^{S} b_i B^i$, then $n \trianglelefteq n'$ stands for $a_i \leqq b_i$ for all $i = 0, \ldots, S$. This can be defined with exponentiation.
2. The variable W_j encodes the values of register R_j for all timepoints.
3. The variable N_i encodes the sequence $a_0, \ldots, a_S$ where $a_l \in \{0, 1\}$ and $a_l = 1$ iff instruction i is done in step l.

Proof of Lemma 2.69 (4).

Structure

In the following we are stating a finite set of equations involving the variables $B, S, C, W_1, \ldots, W_n, N_1, \ldots, N_m, T$ that are all existentially quantified.

This set of equations is constructed in such a way, that P **terminates when started with empty registers** is equivalent to the existence of values for the variables such that the set of equations has a solution. Exponentiation is important as we have to simulate the working of the **GOTO** program which can have an exponential number of configurations.

Clearly, if we had available a single diophantine set which can simulate exponential behaviour (so as to represent all configurations), we could use it and would not need exponentiation.

Proof of Lemma 2.69 (5).

Equations (1)

In order to store the values of the registers as a number in a B representation, we need B to be large enough. For convenience we make B a power of 2 (multiplying a number with B is then just shuffling the entries by one).

$$B > S + 1, \quad B > m, \quad B = 2^C$$

Proof of Lemma 2.69 (6).

Equations (2)

The first eqution is to have available a number T with the value $\sum_{i=0}^{S} B^i$. The second equation makes sure that all entries of N_i are 0's or 1's. The third ensures that only one instruction is active at each point in time.

$$\begin{pmatrix} 1 + (B-1)T & = & B^{S+1} \\ N_i & \trianglelefteq & T \\ N_1 + \ldots + N_m & \trianglelefteq & T \end{pmatrix}$$

Proof of Lemma 2.69 (7).

Equations (3)

Here we ensure that the computation starts with the start state (first equation), ends after step S (second equation), and that the start configuration is with all registers empty (third equation: $B^{S+1} - B = 0 + \sum_{i=1}^{S}(B-1)B^i$).

$$\begin{pmatrix} 1 & \trianglelefteq & N_1 \\ B^S & \trianglelefteq & N_m \\ W_j & \trianglelefteq & B^{S+1} - B \end{pmatrix}$$

Proof of Lemma 2.69 (8).

Equations (4)

Finally we have to **simulate the program using equations**. The first equation is for $i:$ **goto** l, the next two are for $i:\ x_j = x_j + 1$ (for incrementing we define $I = \{1 \leqslant i \leqslant m :\ i:\ x_j = x_j + 1\}$ for decrementing $D = \{1 \leqslant i \leqslant m :\ i:\ x_j = x_j - 1\}$), and the last two are for $i:\ \mathit{If}\ x_j = 0$ **goto** l.

$$\left(\begin{array}{ccc} BN_i & \trianglelefteq & N_l \\ BN_i & \trianglelefteq & N_{i+1} \\ W_j & = & B(B^j + \sum_{i\in I} N_i - \sum_{i\in D} N_i \\ BN_i & \trianglelefteq & N_{i+1} + N_l \\ BN_i & \trianglelefteq & N_{i+1} + BT - 2W_j \end{array}\right)$$

$\rightsquigarrow$ **blackboard 2.8**

Theorem 2.70 (Matiyasevich (1970))

Diophantine equations over $\mathbb{N}$ with exponentiation can be reduced to diophantine equations over $\mathbb{N}$ without it.

Proof.

It suffices to show the existence of one diophantine set D with

- $\langle u, v \rangle \in D$ implies $v \leqslant u^u$,
- for each $k \in \mathbb{N}$ there is $\langle u, v \rangle \in D$ with $v \gneqq u^k$.

This allows us to describe the exponentially many configurations in the simulation of the **GOTO** program.
Matiyasevich proved this ingeniously with Fibonacci numbers:

$$\frac{5}{4}^n \lneqq a_n \lneqq 2^{n-1} \text{ for } n \geqslant 3$$

Corollary 2.71 (Hilbert's 10th Problem is Undecidable)

1. *Hilbert's 10th Problem is* undecidable*.*
2. *All* re *sets are* diophantine *(and vice versa).*
3. *There is a polynomial p in $x_1, \ldots, x_k$ such that the* positive values of p are exactly the prime numbers*. In fact, for each* ***re*** *set there is such a polynomial.*

Using a universal Turing machine, there is indeed **a particular polynomial** $p(x, x_1, \ldots, x_l)$ such that there is no algorithm to decide, given n, whether $p(n, x_1, \ldots, x_l) = 0$ has a solution.

Proof.

1. is obvious.
2. We use the same set of equations as in the proof of Lemma 2.69. The only bit we have to change is to make sure that the machine is not started with empty registers, but with x contained in the first one. We only have to change the third equation on Slide 381 into the following two:

$$\begin{array}{lcl} W_1 & \trianglelefteq & x + B^{S+1} - B, \\ x & \trianglelefteq & W_1. \end{array}$$

3. We use the polynomial $p(x, \overline{y})$ of the diophantine equation and transform it into $q(x, \overline{y}) = x(1 - p^2(x, \overline{y}))$.

2.7 Theorem of Rice

Content of this section (1):

1. We introduce a powerful method for proving that certain problems are undecidable: the Theorem of Rice.
2. We already know that it is undecidable whether a language (given as a Turing machine) is empty, finite, regular, contextfree etc.
3. Rice's theorem states that any nontrivial property of a language is undecidable.
4. Greibach's theorem is an analogue of Rice for language classes and grammars.
5. With Greibach's theorem, it is easy to show directly, that it is undecidable whether a given contextfree grammar is regular, inherently unique (or similar properties).

Content of this section (2):

1. The **smn-Theorem** states that for each partial recursive function $g(x, y)$ there is an algorithm that computes for each DTM for g and x another DTM, that computes $g(x, y)$ upon input of y.
2. The **Recursion theorem** tells us that **there is always a fixed point** for total recursive functions that map indices to indices.

- The following results are independent of a particular gödelization.
- Obviously, any gödelization assigns to each nondeterministic DTM a natural number. But often, **not all numbers encode a nondeterministic DTM**.
- To deal with the last point, we introduce Definition 2.72.
- We use the notation $\mathrm{M}(x_1, \ldots, x_n) \uparrow$ to stand for "TM M, started on the input $x_1\# \ldots \#x_n$ does not terminate".
- We use $\mathrm{M}(x_1, \ldots, x_n) \downarrow y$ to stand for "TM M, started on the input $x_1\# \ldots \#x_n$ terminates with output y". If the output does not matter, we simply write $\mathrm{M}(x_1, \ldots, x_n) \downarrow$.

Definition 2.72 (Gödelization, Index)

Let M_∞ be a fixed DTM which does not terminate (on any input). For a particular **gödelization** $code()$ we define for $e \in \mathbb{N}$

$$\mathrm{M}_e := \begin{cases} \mathrm{M}, & \text{if } code(\mathrm{M}) = e; \\ \mathrm{M}_\infty, & \text{otherwise.} \end{cases}$$

For $e \in \mathbb{N}$ and $n \in \mathbb{N}$, we define the partial recursive function (also called **Gödel numbering**) $\varphi_e^{(n)} : \mathbb{N}^n \to \mathbb{N}$

$$\varphi_e^{(n)}(x_1, \ldots, x_n) := \begin{cases} y, & \text{if } \mathrm{M}_e(x_1, \ldots, x_n) \downarrow y; \\ undef, & \text{if } \mathrm{M}_e(x_1, \ldots, x_n) \uparrow. \end{cases}$$

Wlog, we denote by M_0 a DTM which does never terminate (like M_∞). Similarly, we denote by $\varphi_0^{(n)}$ the partial function on n arguments which is always undefined.

$\varphi_e^{(n)}$ and $W_e^{(n)}$

- Each partial recursive function $f : \mathbb{N}^n \to \mathbb{N}$ has the form $\varphi_e^{(n)}$ for some $e \in \mathbb{N}$. For $n = 1$ we often write φ_e.
- The sets $W_e^{(n)} := Def(\varphi_e^{(n)})$ constitute an enumeration of all re sets of $\mathbb{N}^n$.
- **Is the following function computable?**

$$u : \mathbb{N}^2 \to \mathbb{N},\ (i, n) \mapsto \varphi_i^{(1)}(n)$$

- Is there a total recursive function $h : \mathbb{N}^2 \to \mathbb{N}$ with $\varphi_{h(i,j)}^{(1)} = \varphi_i^{(1)} \circ \varphi_j^{(1)}$?

  This leads to the **smn theorem**.

Step function T_n

- We also use the step function $T_i : \mathbb{N} \to \mathbb{N}$ which assigns to $n \in \mathbb{N}$ the number of steps it takes until the DTM with gödelnumber i, M_i, terminates on input n. If the DTM does not terminate on n, then $T_i(n)$ is undefined. Thus T_i is a partial recursive function.
- The set $\{\langle i, n, t\rangle : T_i(n) = t\}$ is recursive.
- The set $\{\langle i, n, j\rangle : \varphi_i(n) = j\}$ is re but not recursive.
- The set $H_g = \{\langle i, n\rangle : \varphi_i(n) \downarrow\}$ is re but not recursive.
- The set $H = \{i : \varphi_i(i) \downarrow\}$ is re but not recursive.

- What about $Mon = \{i : \forall j \ (\varphi_i(j) \lneqq \varphi_i(j+1))\}$?
- What about $Dec = \{i : \forall j \ \varphi_i(j) \in \{0, 1\}\}$?
- What about $Eq = \{\langle i, j\rangle : \varphi_i = \varphi_j\}$?

Theorem 2.73 (smn Theorem)

For each $m, n \in \mathbb{N}$ *there are* **total recursive functions** $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ *such that for all* $e \in \mathbb{N}$, $\overline{x} \in \mathbb{N}^m$, $\overline{y} \in \mathbb{N}^n$:

$$\varphi_e^{(m+n)}(\overline{x}, \overline{y}) = \varphi_{s_n^m(e,\overline{x})}^{(n)}(\overline{y}).$$

Proof.

Given e and $\overline{x}$, we do the following:

1. Construct new DTM M': it gets n arguments, shuffles them to positions $m+1, \ldots, m+n$ and puts $\overline{x}$ on the first m arguments.
2. It starts M_e (constructed from e) on the $m+n$ arguments.
3. Gödelnumber of the new machine = $s_n^m(e, \overline{x})$.

Definition 2.74 (Index)

Let L be a **re** language. Then there are **denumerable many** DTM's which accept L. The gödelnumbers of them form the index of L.

Let $\mathcal{S}$ be any set of **re** languages. $\mathcal{S}$ is also called property of **re** sets.

Then $I(\mathcal{S}) := \{e : M_e$ accepts a language $L \in \mathcal{S}\}$ is called the index set of $\mathcal{S}$.

Instead of **re** languages, one can also consider as set $\mathcal{S}$

- a set $\mathbf{R} \subseteq \mathcal{R}^{part}$ of partial recursive functions. Then the index set is defined as follows: $I(\mathbf{R}) := \{e : \varphi_e \in \mathbf{R}\}$. Note that $|I(\mathbf{R})| = \infty$, even if $\mathbf{R}$ is finite.

Theorem 2.75 (Rice)

*Let $\mathcal{S}$ be any property of of **re** languages (resp. of partial recursive functions): $\mathcal{S} \subseteq$ type-0.*

If $\mathcal{S}$ is nontrivial *(i.e. $\emptyset \neq \mathcal{S} \neq$ type-0), then the* index set $I(\mathcal{S})$ is undecidable.

Instead of “gödelnumber for M” we also say “index of M”: for each DTM this is uniquely defined.

Proof (of Theorem 2.75).

The nontriviality assumption gives us the existence of L, L' with $L \in \mathcal{S}$, $L' \notin \mathcal{S}$. There are $e, e' \in \mathbb{N}$ with $e \in I(\mathcal{S})$, $e' \notin I(\mathcal{S})$.

We reduce the Halting problem $H = \{i : \varphi_i(i) \downarrow\}$ **or** the complement of it, $\overline{H} = \{i : \varphi_i(i) \uparrow\}$, to $I(\mathcal{S})$.

Case 1: $I(\mathcal{S})$ **does not contain** an index corresponding to M_∞. We show that there is a (total) computable function f with

$$x \in \{i : \varphi_i(i) \downarrow\} \ \underline{\textbf{iff}} \ f(x) \in I(\mathcal{S}) \quad (1)$$

Case 2: $I(\mathcal{S})$ **does contain** an index corresponding to M_∞. We show that there is a (total) computable function f' with

$$x \in \{i : \varphi_i(i) \uparrow\} \ \underline{\textbf{iff}} \ f'(x) \in I(\mathcal{S}) \quad (2)$$

Proof (of Theorem 2.75) (Case 1).

We define

$$g(x,y) := \begin{cases} \varphi_e(y), & \text{for } \varphi_x(x)\downarrow; \\ undef, & \text{else.} \end{cases}$$

g **is computable**: $\exists a \in \mathbb{N}$ s.t. $g(x,y) = \varphi_a^{(2)}(x,y)$.
smn theorem: We get a total recursive function $s_1^2(a,x)$ with $g(x,y) = \varphi_{s_1^2(a,x)}(y)$. Let $f(x) = s_1^2(a,x)$ in the first equivalence on Slide 396.

$\Rightarrow$: Let $\varphi_x(x)\downarrow$. Then $\varphi_{f(x)}(y) = \varphi_{s_1^2(a,x)}(y) = \varphi_e(y)$. Because $e \in I(\mathcal{S})$: $f(x) \in I(\mathcal{S})$.

$\Leftarrow$: Let $\varphi_x(x)\uparrow$. Then $\varphi_{f(x)}(y) = \varphi_{s_1^2(a,x)}(y)$ is the function computed by M_∞. But then $f(x) \notin I(\mathcal{S})$, because we are in **Case 1**.

Proof (of Theorem 2.75) (Case 2).

We define

$$g(x,y) := \begin{cases} \varphi_{e'}(y), & \text{for } \varphi_x(x)\downarrow; \\ undef, & \text{else.} \end{cases}$$

g **is computable**: $\exists a \in \mathbb{N}$ s.t. $g(x,y) = \varphi_a^{(2)}(x,y)$.
smn theorem: We get a total recursive function $s_1^2(a,x)$ with $g(x,y) = \varphi_{s_1^2(a,x)}(y)$. Let $f'(x) = s_1^2(a,x)$ in the second equivalence on Slide 396.

$\Rightarrow$: Let $\varphi_x(x)\downarrow$. Then $\varphi_{f'(x)}(y) = \varphi_{s_1^2(a,x)}(y) = \varphi_{e'}(y)$. Because $e' \notin I(\mathcal{S})$: $f(x) \notin I(\mathcal{S})$.

$\Leftarrow$: Let $\varphi_x(x)\uparrow$. Then $\varphi_{f(x)}(y) = \varphi_{s_1^2(a,x)}(y)$ is the function computed by M_∞. But then $f(x) \in I(\mathcal{S})$, because we are in **Case 2**.

Applications of Rice

Example 2.76 (Using Rice's Theorem)

Apply the theorem of Rice and state $\mathcal{S}$ and $I(\mathcal{S})$ explicitly.

1. Does M accept a **regular** language?
2. Does M terminate on all **inputs of even length**?
3. Does M terminate on each input?
4. Does M ever print "HAPPY-2017" during its computation on tape? $\rightsquigarrow$ **exercise**
5. Is "HAPPY-2017" accepted by M?
6. Does M accept a language that can be also accepted with a DTM M' the number of states of which is a prime number?
7. Does M visit the start state 7 times?

$\rightsquigarrow$ **blackboard 2.9**

- What if we take as set $\mathcal{S}$ **an arbitrary set $\mathbf{M}$ of Turing machines**. And we define the index set as follows:

$$I(\mathbf{M}) := \{e : \mathrm{M}_e \in \mathbf{M}\}?$$

Does that make sense?

- Can one apply Rice's theorem to **Does M run in polynomial time on each input?** What do we get?
- Is the problem "Does M accept L" decidable for particular languages L? What if L itself is decidable?

$\rightsquigarrow$ **blackboard 2.10**

Colloquial formulation of Theorem of Rice

Nontrivial properties of DTM's are undecidable.

What about the following properties:

- DTM has 17 states,
- DTM never ever moves the head to the right.

Beware!

Not all properties of DTM's can be represented as index sets! Not all sets $A \subseteq \mathbb{N}$ are index sets.

Rice is about properties of the language of a TM, not about the TM itself or its behaviour.

Other problems

Correctness: Let a function f be given. Is it decidable, whether a DTM M computes f?

Special case: f corresponds to M_∞.

Specification: Given a specification for a program to be developed. Is it decidable, whether a program satisfies it?

Equivalence: Is it decidable whether two programs compute the same function?

Example 2.77 (NP and P)

Let L be given by a grammar or a DTM.

1. Is L acceptable in polynomial time?
2. Is L decidable in polynomial time?

- Assume **we know that L is in NP**. Is the problem "$L \in \mathrm{P}$?" decidable or not?
- What if **we know that L is in PSPACE** and then ask whether "L can be decided in deterministic logarithmic time"?

- The theorem of Rice helps us to determine that an index set is not recursive.
- Is there an analogue to show that an index set is not even re?

Theorem 2.78 (When is an Index Set not even re?)

*Let $\mathcal{S}$ be a property of recursively enumerable languages. Then the index set $I(\mathcal{S})$ is re, <u>**iff**</u> the following properties are true*

1. *If $L \in \mathcal{S}$ and $L \subseteq L'$ for some **re** L', then $L' \in \mathcal{S}$.*
2. *If L is an infinite set in $\mathcal{S}$, then there is a finite subset L' of L with $L' \in \mathcal{S}$.*
3. *The set of finite languages in $\mathcal{S}$ is **re**.*

Applications

Example 2.79 (Showing an Index set to be re)

Which of the properties of **re** languages L are **re**?

- $L = \Sigma^*$.
- L contains at most 17 elements.
- L contains exactly 17 elements.
- L contains at least 17 elements.
- $L \neq \emptyset$.
- L is not recursive.
- L is not **re**.

$\leadsto$ **blackboard 2.11**

A class $\mathcal{C}$ of languages is **effectively closed** under concatenation with regular sets (resp. union), if RL, LR and $L \cup L'$ can be effectively constructed from R, L, L'.

Theorem 2.80 (Greibach)

Let $\mathcal{C}$ be a class of languages which is **effectively closed** *under concatenation with regular sets and under union. We also assume that the* **check for equality** *with Σ^* is* **undecidable** *for sufficiently large Σ. Let a nontrivial property $\mathcal{S}$ be given, which is (1)* **closed under** $/a$ *($L \in \mathcal{S}$ implies $L/a := \{w : wa \in L\} \in \mathcal{S}$), and (2)* **true for all regular languages**.

Under these assumptions $\mathcal{S}$ is undecidable for $\mathcal{C}$.

Typical application: class of languages = contextfree languages. For a cf grammar G, it is undecidable whether $L(G)$ is regular (but we knew that already).

TU Clausthal
Clausthal University of Technology

2.8 Recursion theorem

A simple (very destructive) game

- Player 1 writes programs, Player 2 manipulates them.
- Player 1 wins, when she can write a program that behaves identical to its manipulated version.
- Would you like to be Player 1 or 2?

Theorem 2.81 (Recursion Theorem)

For each total recursive function $\sigma : \mathbb{N} \rightarrow \mathbb{N}$ *there is* $n_0 \in \mathbb{N}$ *(which can be effectively constructed) with*

$$\varphi_{n_0}(m) = \varphi_{\sigma(n_0)}(m) \text{ for all } m \in \mathbb{N}.$$

Manipulate any TM in an **algorithmic** way: obtain M' from M. Then there is a DTM M_0 which computes the same as M_0': **the manipulation was useless**.

Can you construct a DTM that writes its own index on tape (on empty input) and stops?

Corollary 2.82 (Recursion Theorem: Kleene's version)

To each partial recursive function $\rho : \mathbb{N}^2 \to \mathbb{N}$ *there is* $n_0 \in \mathbb{N}$ *(which can be effectively constructed) with*

$$\varphi_{n_0}(m) = \rho(n_0, m) \text{ for all } m \in \mathbb{N}.$$

Proof.

We consider ρ as a function on $\mathbb{N}$ by fixing the first argument: for each x, the function $\rho(x, y)$ is a (partial) recursive function in y, so it has the form $\varphi_a(\cdot) = \rho(x, \cdot)$, therefore, by the smn theorem, there is a total recursive function $f : \mathbb{N} \to \mathbb{N}$ with $\varphi_{f(x)}(\cdot) = \rho(x, \cdot)$.

We apply the Recursion theorem and get the result. □

Proof (of Theorem 2.81).

We define

$$g(y,x) := \begin{cases} \varphi_z(x), & \text{if } \varphi_y(y) \downarrow z\,; \\ undef, & \text{else.} \end{cases}$$

g **is computable**: $\exists a \in \mathbb{N}$ s.t. $g(x,y) = \varphi_a^{(2)}(x,y)$.
smn theorem: We get a total recursive function $s_1^2(a,x)$ with $g(x,y) = \varphi_{s_1^2(a,x)}(y)$. Let $f(x) = s_1^2(a,x)$, which is total recursive. Thus:

$$\varphi_{f(y)}(x) = \begin{cases} \varphi_{\varphi_y(y)}(x), & \text{if } \varphi_y(y) \downarrow; \\ undef, & \text{else.} \end{cases}$$

Proof (of Theorem 2.81) (2).

We define $h : \mathbb{N} \to \mathbb{N}$, $h(y) := \sigma(f(y))$, which is total recursive. Thus there is i_0 with $h = \varphi_{i_0}$.Therefore $\varphi_{i_0}(i_0)\downarrow$. We get

$$\varphi_{f(i_0)}(x) = \varphi_{\varphi_{i_0}(i_0)}(x) = \varphi_{\sigma(f(i_0))}(x)$$

Therefore it suffices to set $e := f(i_0)$. □

- **Kleene's version says: each DTM can be modified so as to get its own description (index) during a computation and deal with it.**
- Suppose you are working on a DTM M to compute a function $t : \mathbb{N} \rightarrow \mathbb{N}$. But you need to assume that you have available an index of that very machine you are constructing.
- So your DTM is really M' that takes two inputs: $M'(n, m)$ but computes $t(m)$, **asssuming that n is an index of a DTM for t.**
- Kleene's version assures there is indeed such a n_0 and your assumption for using $M'(n_0, m)$ is ok.
- So you can define a macro "get your own description" and use that for any DTM.

Applications of the Recursion Theorem

1. In *any* enumeration $M_1, \ldots, M_i, \ldots$ of *all* DTM's, there are two consecutive DTM's which compute the same function. $\rightsquigarrow$ **exercise**
2. There is e with $\varphi_e(e)\downarrow$ and $\varphi_e(i)\uparrow$ for all $i \neq e$.
3. There is a DTM that outputs its own index.
4. Programs that print out their own source code are also called *quines* (after Willard van Orman Quine). Here are two for Python3 and Ruby:

```
a="a=%c%s%c;print(a%%(34,a,34))";print(a%(34,a,34))
```

```
eval s=%q(puts"eval s=%q(#s)")
```

Applications of the Recursion Theorem (2)

- While the halting problem is undecidable, we may **collect info about particular DTM's** and whether they halt or not.
- Assume we consider a formal system (eg. set theory) in which we can formulate and reason about DTMs. We add more and more axioms about termination of particular DTM's.
- Can we ever reach a state of the formal system that answers all questions about DTMs?
- We assume that such a formal system is correct: it contains and proves only correct statements.
- In particular, if a TM stops after finitely many steps, the system can prove that (it simulates it). If it does not stop, the system does not claim it does.

Lemma 2.83 (No FS captures the whole truth)

For any **consistent and correct formal system** FS *there is a DTM* M *and an input* w_0 *such that*

1. *there is no proof in* FS *that* $M(w_0)$ *terminates,*
2. *there is no proof in* FS *that* $M(w_0)$ *does not terminate.*

Proof.

Let $\rho : \mathbb{N}^2 \to \mathbb{N}$ be the partial recursive (why?) function

$$\rho(i,j) := \begin{cases} 1, & \text{if FS proves that } \varphi_i(j) \uparrow \\ \text{undef}, & \text{else.} \end{cases}$$

By Kleene there is n_0 with $\varphi_{n_0}(j) = \rho(n_0, j)$. Now $\rho(n_0, j) = 1$ **iff** $\varphi_{n_0}(j)\uparrow$. Therefore $\rho(n_0, j)$ must be undefined for all j, **but FS can't prove it.** □

2.9 Oracle TMn

Content of this section:

1. An Oracle DTM is a DTM that can ask an oracle and get back an answer that it can use in further computations.
2. Such an oracle can be a problem we want to show to be undecidable. Maybe we can solve the halting problem with such an oracle.
3. An undecidable oracle allows us to compute more undecidable problems. Such machines can also accept non re languages (eg. the complement of the halting problem).
4. This gives us another notion of reduction that is very useful: Turing reducibility,
5. We introduce a hierarchy of undecidable problems.

Motivation (1)

- To prove undecidability of a problem by **reducing an undecidable problem to it** is a powerful technique, see Definition 2.1 on Slide 207: $A \preceq B$.
- However, it needs a computable function to do the reduction and this makes the method more difficult to apply.
- For example, $H \npreceq \overline{H}$. Therefore $\overline{H}$ cannot be shown undecidable through $\preceq$ alone.
- A better notion seems to use a language A as an oracle: this allows to do **more** reductions.
- If we use $\overline{H}$ as an oracle, surely we can decide H and vice versa.

Motivation (2)

But $\leq$ has its advantages: one can prove more (the price is limited applicability).

- Assume A is **re**. **Then $A \leq \overline{A}$ implies that $\overline{A}$ is re too.** Because $y \in \overline{A}$ iff $f(y) \in A$ (for a computable function $f : A \to \overline{A}$) and any enumeration of A (plus the check that such an element has the form $f(y)$) gives an enumeration of $\overline{A}$.
- We already know from Inf. 3 that H is **re** but not recursive and so $\overline{H}$ is not even **re**: therefore $H \not\leq \overline{H}$.
- If A is not **re** and $A \leq B$, then B is not **re**. So we also have $\overline{H} \not\leq H$ This is not true for oracles.

Motivation (3)

Consider the following undecidable problems:

1. $w_0 \in L(M)$,
2. $L(M) = \emptyset$,
3. $L(M) = \Sigma^*$,
4. $L(M)$ is a regular set.

Intuitively, the first problem seems to be the easiest (**just one check**), while (2) and (3) need an *infinite number of checks* in the worst case. The fourth problem seems to be harder than the first three.

Definition 2.84 (Oracle DTM M^A)

Let $A \subseteq \Sigma^*$. A **Turing machine with oracle A**, denoted by M^A, is a 1-tape DTM with 3 **distinguished states** $q_?, q_y, q_n$.

If it reaches the state $q_?$, it **checks whether the string to the left of the head (until the start of the tape) is contained in A**:

Yes: if it is contained in A, the DTM switches into q_y,

No: otherwise the DTM switches into q_n.

If A is recursive, M^A can be simulated by an ordinary DTM.

- We have introduced in Definition 2.1 the notion of a reduction $\leq$ that we extensively used in proving undecidability.
- In the next definition we distinguish between two variants of $\leq$ and introduce a third one: TM reducibility.
- In the rest of this subsection, we only deal with $\leq_{TM}$.
- From Slide 431 on, we compare all three notions.

Definition 2.85 (Reductions)

A set A_1 is reducible to A_2 **iff** there is a DTM-computable function $f : \mathbb{N} \to \mathbb{N}$, such that:

$$\forall n \in \mathbb{N} \quad \big(n \in A_1 \text{ \underline{\textbf{iff}} } f(n) \in A_2\big).$$

- $\preceq_1$: if the function is a bijection, also called one-one.
- $\preceq_m$: if the function is any function, also called many-one.

A_1 is TM-reducible to A_2, written $A_1 \preceq_{TM} A_2$, if A_1 can be decided by a DTM with oracle A_2.

Definition 2.86 (Recursive, re in A)

Let $A \subseteq \Sigma^*$. A language L **is re in** A, if L is accepted by an oracle DTM M^A.

L **is recursive in** A, if the accepting oracle DTM M^A **always terminates** (i.e. M^A is a decider).

Often we denote by $L^A(\mathrm{M})$ the language accepted by machine M with oracle A: $L(\mathrm{M}^A)$. **(The oracle A does not belong to M.)**

- Are all languages *re in the halting problem*?
- Is each language *re for an appropriate A*?

Definition 2.87 (Equivalence wrt. Turing Reducibility)

We say that two languages L_1, L_2 are *equivalent*, **iff** *each is recursive in the other*. This leads to the notion of *degrees of unsolvability*.

Thus L_1 is accepted by a DTM that takes L_2 as an oracle and vice versa.

Definition 2.88 (Hierarchy)

We consider the language $S_1 = \{e : \ L(\mathrm{M}_e) = \emptyset\}$.

S_1 **is not re.** We define the ascending hierarchy:

$$S_{i+1} := \{e : \ L^{S_i}(\mathrm{M}_e) = \emptyset\}$$

By induction, one verifies easily that

$$S_1 \subsetneq S_2 \subsetneq \ldots \subsetneq S_i \subsetneq S_{i+1} \subsetneq \ldots.$$

Theorem 2.89

The following holds:

S_1 *is* **equivalent** *to "*$w \in L(\mathrm{M})$*",*

S_2 *is* **equivalent** *to "*$L(\mathrm{M}) = \Sigma^*$*",*

S_3 *is* **equivalent** *to "*$L(\mathrm{M})$ *is a regular set".*

Proof.

$\rightsquigarrow$ **blackboard 2.12**

Halting Problem for TM's with oracle A

We use the notation H^A to denote the halting set for TM's with oracle A: the set of indices of all TM's with oracle A that terminate on their own gödelnumber.

For a recursive set A, H^A is exactly the classical halting set H. Obviously, for recursive A, B, the sets H^A and H^B are equivalent (wrt. Definition 2.87).

Jump of a language

For a **re** set A, the jump of A, denoted by A', is the set H^A: the set of all gödelnumbers of machines, that terminate on their own gödelnumber using as oracle the set A.

So A, A', A'', $\ldots$ is a series of sets getting more and more complicated/undecidable.

$\emptyset'$ is the classical halting set. $\emptyset''$ is a new halting set H^H obtained by machines that use the classical halting set as an oracle.

Suppose we are given a problem that is undecidable. **Can we prove this by reducing the halting problem to it?**

Or, to put it differently:

Post's problem

Is the halting problem the simplest undecidable problem?

2.10 Reductions

Content of this section:

1. The Oracle DTM just introduced defines a **reduction** from one set (language) to another set (language): $\leq_{TM}$ **Turing reducibility**.
2. We compare this reduction with two more reductions, that can be derived from Definition 2.1: $\leq_1$ **one-one reducibility** and $\leq_m$ **many-one reducibility**.
3. We introduce the most difficult problems wrt. these reductions: $\leq$- **complete problems**.

Properties of TM-Reduction

Let $\overline{\text{SAT}}$ be the problem to determine **whether a given formula in CNF is unsatisfiable**.

Obviously, we have

- SAT $\leq_{TM}$ $\overline{\text{SAT}}$, and
- $\overline{\text{SAT}}$ $\leq_{TM}$ SAT.

So both are equivalent: no one is harder than the other.

- But this sort of equivalence is rather **coarse**.
- We know that SAT is in NP and $\overline{\text{SAT}}$ in $\mathrm{co\text{-}NP}$.
- TM-equivalence does not imply anything about finer complexity measures.

The following properties hold for all reductions introduced in Definition 2.85.

Lemma 2.90 ($\equiv_1, \equiv_m, \equiv_{TM}$)

The reductions $\leq$ from Definition 2.85 are reflexive *and* transitive.

We can therefore define, for all reductions, the equivalence relations $\equiv_1, \equiv_m, \equiv_{TM}$.

We have seen on Slide 432 that SAT $\equiv_{TM}$ $\overline{\text{SAT}}$.

Obviously $H \equiv_{TM} \overline{H}$, but not $H \equiv_m \overline{H}$ (see the discussion on Slide 418).

Definition 2.91 ($\equiv_{rec}$)

A recursive bijection on $\mathbb{N}$ is a recursive function that is also a bijection.We define $\equiv_{rec}$ on $A, B \subseteq \mathbb{N}$:

$A \equiv_{rec} B$ **iff** there is a recursive bijection with $f(A) = B$.

All relations $\equiv_1, \equiv_m, \equiv_{TM}$ are recursively invariant: If $A \equiv_{rec} B$ then $A \equiv_1 B$, $A \equiv_m B$, and $A \equiv_{TM} B$.

An interesting application of the proof of the Cantor-Bernstein theorem shows the following.

Theorem 2.92 (Myhill)

$A \equiv_{rec} B$ ***iff*** $A \equiv_1 B$.

Proof.

$\rightsquigarrow$ **blackboard 2.13**

Lemma 2.93 (Some facts about $\leq_1$ and $\leq_m$)

1. $A \leq_1 B$ *implies* $A \leq_m B$.
2. $A \leq_1 B$ *implies* $\overline{A} \leq_1 \overline{B}$ *(also for* $\leq_m$*).*
3. *(*$A \leq_m B$ *and* B *recursive) implies* A *recursive.*
4. *(*$A \leq_m B$ *and* B *re) implies* A *re.*

Proof.

$\leadsto$ **blackboard 2.14**

- Are there (nonrecursive) sets, that are incomparable wrt $\leq_m$?

Are there most difficult sets?

Definition 2.94 ($\leq_1$, $\leq_m$–completeness)

$A \subseteq \mathbb{N}$ is $\leq_1$-complete (resp. $\leq_m$-complete), if

1. A is re, and
2. for all **re** sets B: $B \leq_1 A$ (resp. $B \leq_m A$).

- $\{\langle i, n \rangle : \varphi_i(n) \downarrow\}$ is $\leq_1$ complete.
- $\{i : \varphi_i(i) \downarrow\}$ is $\leq_m$ complete.

This is no coincidence!

Lemma 2.95 ($\leq_1$ complete = $\leq_m$ complete)

A set A is $\leq_1$ complete ***iff*** *A is $\leq_m$ complete.*

Relation between $\leq_m$ and $\leq_{TM}$

Lemma 2.96

1. *$A \leq_m B$ implies $A \leq_{TM} B$.*
2. *($A \leq_{TM} B$ and B recursive) implies A recursive.*
3. *If $A \leq_m B$ and $B \leq_{TM} C$, then $A \leq_{TM} C$.*

“$A \leq_{TM} B$ implies $A \leq_m B$” is false: (see also Slide 419)

- $\overline{\{\langle i,n\rangle : \varphi_i(n)\downarrow\}} \leq_{TM} \mathbf{S_1}$, but
- $\overline{\{\langle i,n\rangle : \varphi_i(n)\downarrow\}} \not\leq_m \mathbf{S_1}$.

“($A \leq_{TM} B$ and B re) implies A re.” is false:

- $\overline{\{\langle i,n\rangle : \varphi_i(n)\downarrow\}} \leq_{TM} \{\langle i,n\rangle : \varphi_i(n)\downarrow\}$, and
- $\overline{\{\langle i,n\rangle : \varphi_i(n)\downarrow\}}$ is not **re** but can be recognized with oracle $\{\langle i,n\rangle : \varphi_i(n)\downarrow\}$.

We are now coming back to the question on Slide 427.

- How do the equivalence classes of $\equiv_{TM}$ look like? They are called Turing degrees.
- The Turing degrees form a poset, in fact a join semilattice but not a lattice.
- The least element is the set of all recursive sets, denoted by $\mathbf{O}$ (the equivalence class of all recursive sets).
- For a degree $\mathbf{d}$ the jump of $\mathbf{d}$, denoted by $\mathbf{d}'$ is defined as the degree consisting of the equivalence class of A' (the jump defined on Slide 427), where $A \in \mathbf{d}$.

- Thus $\mathbf{O}'$ is the Turing degree of the halting problem. Obviously $\mathbf{d}'$ is not recursive in $\mathbf{d}$, it is more complicated.
- Post asked whether there are any **re** degrees between $\mathbf{O}$ and $\mathbf{O}'$.
- Friedberg and Muchnik solved this problem by inventing the priority method: there are indeed such degrees strictly between $\mathbf{O}$ and $\mathbf{O}'$.
- Thus the halting problem is not the simplest undecidable problem: in fact, there is no simplest undecidable problem.

3. SPACE versus TIME

3 SPACE versus TIME
- Basics
- Some Hierarchies
- Relating TIME and SPACE
- More Properties

Contents of this chapter

- While we investigated in the previous chapters the boundary between decidability and undecidability, in this and the next chapter we consider only decidable languages.
- We show how to measure time and space complexity of a given language.
- Such measurements make only sense up to a constant factor: space compression and linear speed-up. Restricting to 1-DTM's results in a quadratic time increase, while space complexity remains the same.
- We will see: (1) **time-bounded** problems are decidable (trivial), and (2) **space-bounded** problems are **time-bounded** and thus decidable (not so trivial).

Contents of this chapter (2)

- We show several tight, and strict hierarchies for space and time.
- We investigate the precise relations between nondeterministic und deterministic Space- and Time classes, including Savitch's Theorem.
- We show that there are arbitrarily large gaps for space complexity (Borodin's gap theorem) which leads to unintuitive consequences.
- There are also languages that can be sped up indefinitely (Blum's speed up theorem).
- We prove the Union theorem.

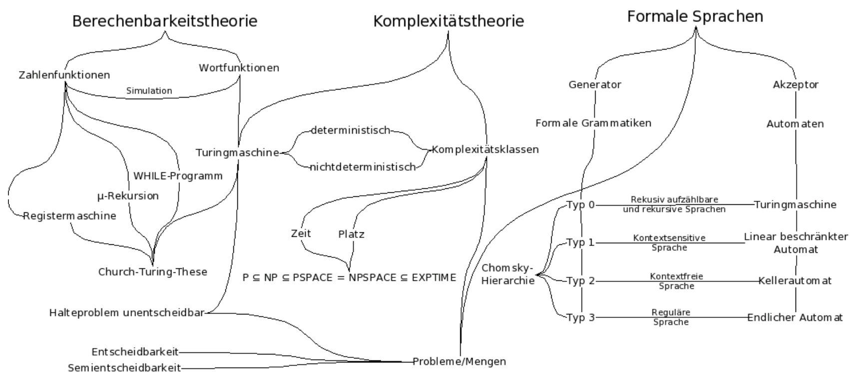

Figure 17: Some relations.

3.1 Basics

Definition 3.1 ($\mathrm{NTIME}(T(n))$, $\mathrm{DTIME}(T(n))$)

Base model: k-DTM M (**one tape for the input**).
If M makes at most $T(n)$ steps for each input of length n, it is called $T(n)$ time-bounded.
The accepted language by M has time complexity $T(n)$ (in fact, we mean $\max(n+1, \lceil T(n) \rceil)$).

- $\mathbf{DTIME}(T(n))$ is the class of languages, that are accepted by $T(n)$ time-bounded, DTM's.
- $\mathbf{NTIME}(T(n))$ is the class of languages, that are accepted by $T(n)$ time-bounded NTM's.

Definition 3.2 ($\mathrm{NSPACE}(S(n))$, $\mathrm{DSPACE}(S(n))$)

Base model: k-DTM M (**one special tape just for the input, input remains there** (offline DTM)).
If M uses at most $S(n)$ cells (on all tapes) for each input of length n, it is called $S(n)$ space-bounded.
The accepted language by M has space complexity $S(n)$ (in fact, we mean $\max(1, \lceil S(n) \rceil)$).

- $\mathbf{DSPACE}(S(n))$ is the class of languages, that are accepted by $S(n)$ space-bounded, DTM's.
- $\mathbf{NSPACE}(S(n))$ is the class of languages, that are accepted by $S(n)$ space-bounded, NTM's.

Question:

Why *offline*-TM's?

Space functions of less than linear growth.

Question:

To which time/space class belongs

$$L_{\text{mirror}} := \{wcw^R : \; w \in (0+1)^*\},$$

all words **mirrored around** c?

Time: $\mathbf{DTIME}(n+1)$. Copy the input to the right of c **in reverse order**.
After finding c, just check the coming part.

Space: The machine described above gives $\mathbf{DSPACE}(\lceil \frac{n}{2} \rceil)$.

Can we do better?

Yes: $\mathbf{L} \in \mathbf{DSPACE}(\lg n)$. Use two tapes as binary counters.
Checking that there is exactly one c can be done with no space. Then we check symbol for symbol on the right and left side. We need the counters to keep track of the positions.
One can easily do with one counter and one tape.

- How **many tapes** are needed?
- What is the influence of the **size of the alphabet**?
- What about the **number of states**?

Theorem 3.3 (Tapes, Alphabet, States)

Let L be a language accepted by an arbitrary TM (no restriction on the number of tapes, number of states, number of tape symbols).

Minimal Alphabet: *L can be accepted by an offline NTM (DTM) with only one tape and two tape symbols $\{\#, 1\}$.*

Small State Space: *L can be accepted by an offline NTM (DTM) with only one tape and only three states.*

Proof

Intuitively, it is obvious that one tape suffices: one can always use tracks (see the proof of Theorem 1.63). Thus instead of one symbol per cell, one can always use new symbols that code finitely many symbols.

To minimize the number of tape symbols, one can certainly simulate with 2 tape symbols a set of finitely many tape symbols (simply as particular sequences). The finite control of the program can then distinguish between the new codes.

To minimize the number of states, the finite control can easily be simulated with finitely many new tape symbols. One needs only 3 states: one for the start state, one for the halt state and a third one to distinguish it from the start state. So there are effectively 2 states (to encode the finite control with the tape symbols), plus the halt state.

Important questions:

Time: Is $\text{DTIME}(f(n))$ contained in the class of recursive languages?

Space: Is $\text{DSPACE}(f(n))$ contained in the class of recursive languages?

Time/Space: What about $\text{NTIME}(f(n))$, $\text{NSPACE}(f(n))$?

Of course, the answers depend on the functions f. We require them to be recursive.

Later, in order to formulate more precise relationships, we need more restricting notions (Time/space constructible).

Only the growth rate in a complexity class counts:
Constant factors can be ignored.

Theorem 3.4 (Tape Compression)

Let L be accepted by a k-DTM (resp. k-NTM) M_1 which is $S(n)$ space-bounded.
Then for any $c > 0$, L is also accepted by a k-DTM (resp. k-NTM) M_2 which is $cS(n)$ space-bounded.

Proof

One direction is trivial. The other is by coding a fixed number r $(> \frac{2}{c})$ of adjacent cells on the tape as one new symbol. **The states of the new machine simulate the movements of the head as state transitions** (within the new symbol). For r cells of the old machine, we only use at most 2: the worst case arises when one is forced to move from one block in the adjacent one.

Corollary 3.5 (Tape Compression)

For all $c > 0$ and space functions $S(n)$:

$$\mathbf{DSPACE}(S(n)) = \mathbf{DSPACE}(cS(n))$$
$$\mathbf{NSPACE}(S(n)) = \mathbf{NSPACE}(cS(n))$$

Theorem 3.6 (Time Speed Up)

Let L be accepted by a k-DTM M_1 which is $T(n)$ **time-bounded**. *We assume that $k > 1$ and* $\inf_{n\to\infty} \frac{T(n)}{n} = \infty$.
Then for any $c > 0$, L is also accepted by a k-DTM M_2 which is $cT(n)$ **time-bounded**.

Proof

The proof is again by coding a fixed number r ($> \frac{22}{c}$) of adjacent cells on the tape as one new symbol. **The states of the new machine simulate the movements of the head as state transitions** (within the new symbol).

- The new machine has to copy and encode the input: n moves.
- And $\lceil \frac{n}{r} \rceil$ steps to print the new symbols encoding the entire input.

Proof (2)

When simulating the old machine, the new one makes 11 instead of r steps:

- It starts with scanning the neighboring cells: one move to the right, two moves to the left, and one to the right. This information is stored in the finite control. 4 moves.
- Simulating the old machine: **This takes no time, as it is built-in to the transition rules.** 0 moves.
- The machine moves only when the heads have to move into neighbouring cells (to change them) or to change the current one. One move to overwrite the current symbol, one to move left, one to overwrite it, two to move right, one to overwrite it, one to move back (in the worst case). 7 moves.

Proof (3)

Thus if the old machine does $T(n)$ moves, the new one does at most

$$n + \lceil \frac{n}{r} \rceil + 11 \lceil \frac{T(n)}{r} \rceil \lneqq n + \frac{n}{r} + 11 \frac{T(n)}{r} + 2$$

steps. **Our assumption $\inf_{n \to \infty} \frac{T(n)}{n} = \infty$ ensures that for each constant d: $n \leqslant \frac{T(n)}{d}$.** Thus we get an upper bound of

$$T(n)\left(\frac{11}{r} + \frac{2}{d} + \frac{1}{rd}\right)$$

It remains to choose d properly: $d = \frac{r}{2}$. This gives an upper bound of $\frac{11}{r} + \frac{4}{r} + \frac{2}{r}$ and together with $r \gneqq \frac{22}{c}$ we get $\frac{11}{22}c + \frac{4}{22}c + \frac{2}{22}c = \frac{17}{22}c$ which is $\lneqq c$.

Corollary 3.7 (Time Speed Up)

For each $c > 0$ *and time functions* $T(n)$ *with* $\inf_{n\rightarrow\infty} \frac{T(n)}{n} = \infty$*:*

$$\mathbf{DTIME}(T(n)) = \mathbf{DTIME}(cT(n))$$
$$\mathbf{NTIME}(T(n)) = \mathbf{NTIME}(cT(n))$$

Question:

What happens if less tapes are available?

Consider the language L_{mirror}: **Linear complexity does not hold any more, if there is only one tape.**

However, we have

Theorem 3.8 (Tape Reduction for TIME)

1. *If $L \in \mathbf{DTIME}(T(n))$, then L is accepted by a*
   - 1-DTM *that is time-bounded by $T^2(n)$,*
   - 2-DTM *that is time-bounded by $T(n) \lg T(n)$.*
2. *If $L \in \mathbf{NTIME}(T(n))$, then L is accepted by a*
   - 1-NTM *that is time-bounded by $T^2(n)$,*
   - 2-NTM *that is time-bounded by $T(n) \lg T(n)$.*

Proof

How many steps are needed to simulate a k-DTM with a 1-DTM?

Consider $L := \{ww^R : w \in \Sigma^*\}$.

1-DTM: How to decide L with a 1-DTM? I.e. for m steps of a k-DTM, a 1-DTM uses m^2 many steps (to be precise, $6m^2$).
We speed up by a factor of $\frac{1}{\sqrt{6}}$.

2-DTM: How to decide L with a 2-DTM? Here we need a counter to count up to n: this gives a factor of just $\lg T(n)$, not $T(n)$.

The last theorem is also true for space-bounded functions.

Theorem 3.9 (Tape Reduction for SPACE)

- *If* $L \in \mathbf{DSPACE}(S(n))$*, then* L *is accepted by a* 1-DTM *that is space-bounded by* $S(n)$*.*
- *If* $L \in \mathbf{NSPACE}(S(n))$*, then* L *is accepted by a* 1-NTM *that is space-bounded by* $S(n)$*.*

Proof.

Proof like the last one. While simulating a k-DTM with a 1-DTM one needs the same number of cells: no speed-up is needed (we simply use more tape symbols). Compare with the proof of Theorem 3.3. □

Nondeterministic computations are **shorter** than deterministic ones. But each NTM can be **simulated** by a DTM. The following lemma gives the precise relation.

Lemma 3.10 (NTIME vs. DTIME)

If $L \in \mathbf{NTIME}(f(n))$*, then there is* $c > 0$ *s.t.* $L \in \mathbf{DTIME}(c^{f(n)})$.

(Proof of Theorem 3.10 (1)).

It is obvious how to simulate a NTM with a DTM: simply **enumerate all possible paths**. However, to get the precise bound used above, we **compute the number of ID's for an input of length n**. Assume the NTM M has s states, k tapes, the alphabet is based on t symbols and the input is of length n. Then in time $f(n)$ at most $f(n)$ tape cells can be visited. Thus we have at most $s(f(n)+1)^k t^{kf(n)}$ many different ID's. For $d := s(t+1)^{3k}$ we get for all $n \in \mathbb{N}$ a simpler upper bound

$$s(f(n)+1)^k t^{kf(n)} \leqslant d^{f(n)}.$$

(Proof of Theorem 3.10 (2)).

How can a DTM determine whether M accepts an input of length n?

It constructs a list of all ID's accessible from the initial ID.

How much time is needed to do this?

It is bounded by the (length of the list)^2.

There are at most $d^{f(n)}$ ID's, each can be encoded with $1 + k(f(n) + 1)$ symbols. Thus the length of the list is at most $d^{f(n)}(1 + k(f(n) + 1))$. This gives a time bound of the multitape DTM of $\left(d^{f(n)}(1 + k(f(n) + 1))\right)^2$. And this is bounded by $c^{f(n)}$ for a suitable c. □

3.2 Some Hierarchies

Theorem 3.11 ($\mathbf{DSPACE}(f(n)) \subsetneq$ Rec)

Let $f(n)$ be any total recursive function. Then there is a recursive language L that is not in $\mathbf{DTIME}(f(n))$ (resp. not in $\mathbf{DSPACE}(f(n))$).

Proof.

We use diagonalization. Let M be the DTM computing $f(n)$. Let $\mathrm{M}_1, \dots, \mathrm{M}_i, \dots$ be an enumeration of all DTM's. Let $x_1, \dots, x_i, \dots$ be an enumeration of all words of $\{a, b\}^*$. Define the language

$$L = \{x_i : \ \mathrm{M}_i \text{ does not accept } x_i \text{ within } f(|\, x_i \,|) \text{ moves}\}$$

L is recursive but not in $\mathbf{DTIME}(f(n))$.
The proof for $\mathbf{DSPACE}(f(n))$ is similar. □

Well-behaved Functions

In the remainder of this chapter we state several relations between complexity classes. Unfortunately, these do not hold for arbitrary space or time functions. But they do hold for almost all naturally occurring functions.

The technical notions we need to assume are:

Space: space functions $S(n)$ need to be (fully) space constructible: for each n there is an input of length n s.t. a TM uses on this input exactly $S(n)$ cells.

Time: time functions $T(n)$ need to be (fully) time constructible: for each n there is an input of length n s.t. a TM runs exactly $S(n)$ steps on this input.

In the following, remember our assumption on space/time functions (Definitions 3.1, 3.2).

Definition 3.12 ((Fully) Space Constructible)

A function $S : \mathbb{N} \rightarrow \mathbb{N}$ is fully space constructible, if there is a $S(n)$ space-bounded Turing machine M such that on each input of length n, M uses exactly $S(n)$ cells.
S is called space constructible, if there is a $S(n)$ space-bounded Turing machine M such that on some input of length n, M uses exactly $S(n)$ cells.

Definition 3.13 ((Fully) Time Constructible)

A function $T : \mathbb{N} \rightarrow \mathbb{N}$ is fully time constructible, if there is a $T(n)$ time-bounded Turing machine M such that on each input of length n, M runs exactly $T(n)$ steps.
S is called time constructible, if there is a $T(n)$ time-bounded Turing machine M such that on some input of length n, M runs exactly $T(n)$ steps.

Lemma 3.14

We consider space functions $S(n)$.

1. *Each space constructible function $S(n)$ with $S(n) \geqslant n$ is also* **fully space constructible**.
2. *If a language L is accepted by a $S(n)$ space-bounded TM with $S(n) \geqslant \lg(n)$, then L is accepted by a $S(n)$ space-bounded TM* **that halts on all inputs**.

Proof.

$\rightsquigarrow$ **exercise**

Definition 3.15 (Well-behaved Functions)

Consider the following space of functions from $\mathbb{N}$ in $\mathbb{N}$. It contains $\lceil \log_a n \rceil$ $(a \in \mathbb{N})$, n^k $(k \in \mathbb{N})$, 2^n, $n!$ and is closed wrt. *multiplication*, *exponentiation*, *composition* and multiplication by k $(k \in \mathbb{N})$. We call such functions well-behaved.

Lemma 3.16

Well-behaved functions f *with* $f(n) \geq \lg(n)$ *are* space constructible.

Proof.

$\rightsquigarrow$ **exercise**

Theorem 3.17 (Time/Space Hierarchies)

Let $S_2(n)$ be space constructible and $T_2(n)$ fully time constructible. We also assume, that $S_1(n) \geqslant \lg n$.

1. *If* $\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$ *then* $\mathbf{DSPACE}(S_1(n)) \subsetneqq \mathbf{DSPACE}(S_2(n))$.
2. *If* $\inf_{n\to\infty} \frac{T_1(n)\lg T_1(n)}{T_2(n)} = 0$ *then* $\mathbf{DTIME}(T_1(n)) \subsetneqq \mathbf{DTIME}(T_2(n))$.

Proof of Theorem 3.17

First we **assume that S_2 is fully space constructible** to simplify the proof. The general version will be done in the lab.
We fix an enumeration $M_1, \ldots, M_i, \ldots$ of all offline DTM's over $\{0, 1\}$ with (wlog.) **one** storage tape. **Later we represent these machines as words w over $\{0, 1\}^*$**.

Thus each word w (sequence of 0's and 1's) can be seen as a DTM M_w, s.t. there are infinitely many machines occurring in the enumeration that accept the same language (see last chapter). **We also assume, wlog, that the size of machine M_i in its binary representation is greater than i (adding useless states).** Thus, if machines corresponding to a language L occur infinitely often, their sizes in the binary representation get arbitrarily large (a property we need later).

Proof of Theorem 3.17 (2)

We construct a DTM M that uses $S_2(n)$ space and disagrees on at least one input with any $S_1(n)$ space bounded DTM.

Given a word w, the machine M first marks $S_2(|w|)$ cells on the storage tape (this is where the fully space constructible assumption comes in). Whenever the machine is about to move beyond these bounds, it stops and rejects the input.

Then M simulates M_w, the TM encoded by w. **If M_w is $S_1(n)$ space bounded (and uses t tape symbols), what is the space required to do this simulation?**

M needs $\lceil \lg(t) \rceil S_1(n)$ space.

Proof of Theorem 3.17 (3)

Now we define M **accepts** w **iff** (1) M can do the simulation in time $S_2(n)$ and (2) M_w stops **without accepting** w.
Obviously, $L(M)$ is in $\mathbf{DSPACE}(S_2(n))$.

To show that $L(M)$ is not in $\mathbf{DSPACE}(S_1(n))$, we assume it is and show a contradiction. By Lemma 3.14, there is a DTM $\hat{M}$ accepting $L(M)$, working in space $S_1(n)$ and always terminating. Assume it is using t tape symbols, so t is fixed.
$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$ guarantees that there is a large enough w, such that $\lceil \lg(t) \rceil S_1(|w|) \lneq S_2(|w|)$ and $L(M_w) = L(\hat{M}) = L(M)$ ($L(\hat{M})$ occurs infinitely often in the enumeration: our assumption $|w_i| > i$ ensures the inequation).

Thus $\hat{M}$ can do the simulation and accepts w **iff** M_w does not accept w, which is a contradiction.

For the second part of the theorem $\rightsquigarrow$ **blackboard 3.1**.

With the last theorem, we get:

$$\mathbf{DSPACE}(n) \subsetneqq \mathbf{DSPACE}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{DSPACE}(n^r) \subsetneqq \ldots$$

and

$$\mathbf{DTIME}(n) \subsetneqq \mathbf{DTIME}(n^2) \subsetneqq \ldots \subsetneqq \mathbf{DTIME}(n^r) \subsetneqq \ldots$$

as well as

$$\mathbf{DSPACE}(\lg n) \subsetneqq \mathbf{DSPACE}(\lg^2 n) \subsetneqq \ldots \subsetneqq \mathbf{DSPACE}(\lg^r n) \subsetneqq \ldots$$

and

$$\mathbf{DTIME}(\lg n) \subsetneqq \mathbf{DTIME}(\lg^2 n) \subsetneqq \ldots \subsetneqq \mathbf{DTIME}(\lg^r n) \subsetneqq \ldots$$

Question:

How about **nondeterministic** Space and Time?

- The analogue of Theorem 3.17 holds.
- There is no **simple** proof. The result is a corollary to the Theorem of Immerman and Szelepcsenyi.

The proof of the next theorem is straightforward.

Theorem 3.18 (Nondeterministic Hierarchies)

Let $S_1(n), S_2(n)$ *and* $f(n)$ *be fully space constructible. We also assume that* $S_2(n) \geqslant n$ *und* $f(n) \geqslant n$ *for all* $n \in \mathbb{N}$. *Let* $T_1(n), T_2(n)$ *and* $f'(n)$ *be fully time constructible.*

1. $\mathbf{NSPACE}(S_1(n)) \subseteq \mathbf{NSPACE}(S_2(n))$ *implies* $\mathbf{NSPACE}(S_1(f(n))) \subseteq \mathbf{NSPACE}(S_2(f(n)))$.
2. $\mathbf{NTIME}(T_1(n)) \subseteq \mathbf{NTIME}(T_2(n))$ *implies* $\mathbf{NTIME}(T_1(f'(n))) \subseteq \mathbf{NTIME}(T_2(f'(n)))$.

The proof uses a technique called padding that can be applied in many situations (and ensures to get the same results for deterministic complexity classes).

Proof of Theorem 3.18.

Let L_1 be a language accepted by M_1 in nondeterministic space $S_1(f(n))$. **We have to show that L_1 can be accepted in space $S_2(f(n))$, using the hypothesis.**

We associate to L_1 its padded version:

$$L_2 = \{w\S^i : \ \mathrm{M}_1 \text{ accepts } w \text{ in space } S_1(|w| + i)\}.$$

L_2 can be accepted by a TM working in space $S_1(n)$: it marks the $S_1(|w| + i)$ cells (**S_1 is fully constructible**) and simulates M_1 on w. It accepts only if M_1 does **and** M_1 does so **not using more than** $S_1(|w| + i)$ many cells.

By hypothesis, there is M_3 which is $S_2(n)$ space bounded accepting L_2. $\rightsquigarrow$ **blackboard 3.2** □

Applying Theorem 3.18

Assume $\mathbf{NSPACE}(n^4) \subseteq \mathbf{NSPACE}(n^3)$. We apply Theorem 3.18 for n^3, n^4 and n^5 separately:

$$\begin{array}{rcl} \mathbf{NSPACE}(n^{12}) & \subseteq & \mathbf{NSPACE}(n^9) \\ \mathbf{NSPACE}(n^{16}) & \subseteq & \mathbf{NSPACE}(n^{12}) \\ \mathbf{NSPACE}(n^{20}) & \subseteq & \mathbf{NSPACE}(n^{15}) \end{array}$$

So we get:

$$\mathbf{NSPACE}(n^{20}) \subseteq \mathbf{NSPACE}(n^9).$$

Using Theorem 3.21 (coming soon on Slide 487) we know that $\mathbf{NSPACE}(n^9) \subseteq \mathbf{DSPACE}(n^{18})$. We also know that $\mathbf{DSPACE}(n^{18}) \subseteq \mathbf{NSPACE}(n^{18})$ (trivial).
Using Theorem 3.17: $\mathbf{DSPACE}(n^{18}) \subsetneq \mathbf{DSPACE}(n^{20})$, which is a contradiction.

The last theorem implies the following hierarchies:

$$\mathbf{NSPACE}(n) \subsetneq \mathbf{NSPACE}(n^2) \subsetneq \ldots \subsetneq \mathbf{NSPACE}(n^r) \subsetneq \ldots$$

and

$$\mathbf{NTIME}(n) \subsetneq \mathbf{NTIME}(n^2) \subsetneq \ldots \subsetneq \mathbf{NTIME}(n^r) \subsetneq \ldots$$

similarly

$$\mathbf{NSPACE}(\lg n) \subsetneq \mathbf{NSPACE}(\lg^2 n) \subsetneq \ldots \subsetneq \mathbf{NSPACE}(\lg^r n) \subsetneq \ldots$$

and

$$\mathbf{NTIME}(\lg n) \subsetneq \mathbf{NTIME}(\lg^2 n) \subsetneq \ldots \subsetneq \mathbf{NTIME}(\lg^r n) \subsetneq \ldots$$

Theorem 3.19 (Deterministic Hierarchies)

Let $S_1(n), S_2(n)$ and $f(n)$ be fully space constructible. We also assume that $S_2(n) \geqslant n$ and $f(n) \geqslant n$ for all $n \in \mathbb{N}$. Let $T_1(n), T_2(n)$ and $f'(n)$ be fully time constructible.

1. $\mathbf{DSPACE}(S_1(n)) \subseteq \mathbf{DSPACE}(S_2(n))$ *implies* $\mathbf{DSPACE}(S_1(f(n))) \subseteq \mathbf{DSPACE}(S_2(f(n)))$.
2. $\mathbf{DTIME}(T_1(n)) \subseteq \mathbf{DTIME}(T_2(n))$ *implies* $\mathbf{DTIME}(T_1(f'(n))) \subseteq \mathbf{DTIME}(T_2(f'(n)))$.

Proof.

Completely analagous to the proof of Theorem 3.18. □

3.3 Relating TIME and SPACE

Theorem 3.20 (Relationship between TIME and SPACE)

1. $\mathbf{DTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$.
2. *If* $f(n) \geqslant \lg n$ *and* $L \in \mathbf{DSPACE}(f(n))$*, then there is* $c > 0$ *s.t.* $L \in \mathbf{DTIME}(c^{f(n)})$.
3. *If* $L \in \mathbf{NSPACE}(f(n))$*, then there is* $c > 0$ *s.t.* $L \in \mathbf{DTIME}(c^{f(n)})$.

Proof

1. Obvious.

2. Assume there are s states and t tape symbols. Using $f(n)$ cells, the number of ID's for an input of length n is bounded by $(s+1)nf(n)t^{f(n)}$ (using Theorem 3.9 we assume we deal with a *offline*-DTM with one storage tape).
Because of $f(n) \geqslant \lg n$ there is c such that for $n \geqslant 1$:
$c^{f(n)} \geqslant (s+1)nf(n)t^{f(n)}$ (why?).
We use a 3-DTM: One tape to count up to $c^{f(n)}$, the others to simulate the old machine. If no accepting state is reached when the counter is maximal, the machine does not accept. Otherwise the new machine accepts.

Proof (2)

3. Again we can bound the number of ID's by $c^{f(n)}$, but by using a depth-first search after that, we only get a double exponential bound.

In the following we show that the tree can be pruned so that each ID occurs only once. We do this by a recursively defined list.

How many steps are needed to construct this list?

1. l^2 many comparisons (l number of ID's),
2. multiplied with their maximal length ($c' f(n)$).

Altogether a bound of the form $c''^{f(n)}$.

Theorem 3.21 (NSPACE vs. DSPACE (Savitch))

For each fully space constructible function $S(n)$ *:*

$$\mathbf{NSPACE}(S(n)) \subseteq \mathbf{DSPACE}(S^2(n))$$

Proof

- Again there are $c^{S(n)}$ different ID's for an input of length n.
- Their maximal length is $S(n)$. If the input is accepted, it is so in at most $c^{S(n)}$ steps.
- Let $Test(I_1, I_2, i)$ the statement

  I_2 can be reached from I_1 in at most 2^i steps.

- $Test(I_1, I_2, i)$ can be solved recursively, by calling $Test(I_1, I', i-1)$ and $Test(I', I_2, i-1)$ (for all ID's I' of length at most $S(n)$).

Proof (2)

We construct a DTM as follows.

- For the initial configuration I_0 and all accepting configurations I_f (of length $S(n)$), we check whether $Test(I_0, I_f, S(n) \lg c)$ (the third argument is the length of $c^{S(n)}$).
- This goes down recursively until the third argument is 0: $Test(I, I', 0)$ is true **iff** either $I = I'$ or I' is reachable from I in one step.
    - original check is successful

iff

DTM accepts

Proof (3)

The space used by the above DTM?

- There are $O(S(n))$ many calls of $Test$. For each call we have to store two ID's and the counter (third argument).
- The ID's have maximal length $S(n)$ and the counter $\lg S(n)$. So we need storage of $\mathbf{O}(S(n))$.
- With $\mathbf{O}(S(n))$ many calls, we need space of size $\mathbf{O}(S^2(n))$.

The following is a beautiful theorem from Hopcroft, Paul and Valiant from 1975. No serious improvements have been found until today.

Theorem 3.22 ($\mathbf{DTIME}(t(n)) \subseteq \mathbf{DSPACE}(\frac{t(n)}{\lg t(n)})$)

For all time functions $t(n)$:

$$\mathbf{DTIME}(t(n)) \subseteq \mathbf{DSPACE}(\frac{t(n)}{\lg t(n)})$$

In particular, for fully time/space constructible functions, $\mathbf{DTIME}(t(n)) \subsetneq \mathbf{DSPACE}(t(n))$.

The proof on the next slides follows closely the original paper of Hopcroft, Paul and Valiant from 1977.

Proof (0). Main idea.

- We simulate a DTM running in time $t(n)$ by walking through a graph.
- The graph has $t(n)^{\frac{1}{3}}$-many nodes. These nodes are time segments $\triangle$ of length exactly $t(n)^{\frac{2}{3}}$.
- For simulating, we need the content of the tape. By a tricky argument we ensure that only a fraction of the whole content is needed ($\leadsto$ pebble game).
- We have to minimise space, not time. By storing less (but the right) content, we can reconstruct what is needed for the simulation.

Proof (1). Step 1: The pebble game.

We are playing a game by placing **pebbles** on the vertices of such a graph.

The goal is to be able to place a pebble on a particular vertex, determined in advance, before the game starts.

This should be done in such a way, that the maximum number of pebbles in the graph (at any moment in time) is below a certain bound.

In the general case, we must place a pebble on all (but one) nodes. Therefore we need **more structure**: let G_k be the set of all finite, directed, acyclic graphs (dag) with **indegree** at most k. Vertices with indegree 0 are **input** vertices.

$\rightsquigarrow$ **blackboard 3.3**

Proof (2). Step 1: The pebble game. (2)

Rules for placing and moving the pebbles:

1. A pebble can always be placed on an input vertex.
2. If all in-nodes of a vertex v have pebbles, then a pebble can be placed on v (the vertex **fires**).
3. Any pebble can be removed at any time.

Let $P_k(n)$ be the maximum **over all graphs in G_k with n vertices** of the number of pebbles required to place a pebble on an arbitrary vertex of such a graph.

Lemma 3.23

For each k, $P_k(n) = \mathbf{O}(n/\lg n)$.

The bound is tight. $\rightsquigarrow$ **blackboard 3.4**

Proof (3). Step 2: Relating the game with the problem.

We simulate a k-DTM of time complexity $t(n)$ by a NTM of space complexity $\frac{t(n)}{\lg t(n)}$. (This gives us the result for $\mathbf{NSPACE}(\frac{t(n)}{\lg t(n)})$: $\mathbf{DSPACE}(\frac{t(n)}{\lg t(n)})$ will be done separately.)

1. Each tape is partitioned into blocks of size $t(n)^{\frac{2}{3}}$.
2. Modify k-DTM so that it **respects blocks** (add one more tape as a *clock*: blocks are crossed only at multiples of time $t(n)^{\frac{2}{3}}$ (if a crossing is attempted at other times, simulation is stopped). The new machine works still in time $t(n)$: it is only **slower by a constant**.
   What if a head crosses all the time the boundary? Slow down of $t(n)^{\frac{2}{3}}$.
   How to repair? $\rightsquigarrow$ **blackboard 3.5**
3. The new DTM runs in segments Δ of time periods of length $t(n)^{\frac{2}{3}}$. There are $t(n)^{\frac{1}{3}}$ such time segments Δ.

Proof (4). Step 2: Relating the game with the problem (2).

1. For $h(i, \Delta)$ being the position of the head of tape i after segment Δ, we define $h(\Delta) = [h(1, \Delta), \ldots, h(k, \Delta)]$ and $h = [h(1), \ldots, h(t(n)^{\frac{1}{3}})]$.
2. Directed graph: The $t(n)^{\frac{1}{3}}$ vertices $v(\Delta)$ correspond to the time segments Δ of the computation of the DTM. For tape i let Δ_i be the last segment (prior to Δ) where head i scanned the same block (as for Δ).
3. The edges are: from $v(\Delta - 1)$ to $v(\Delta)$ and for $1 \leqslant i \leqslant k$: from $v(\Delta_i)$ to $v(\Delta)$.
4. What is the space needed to describe this graph? It is $(k+1)t(n)^{\frac{1}{3}} \lg t(n)$.

Proof (5). Step 2: Relating the game with the problem (3).

1. We denote by $c(\Delta) = [c(1, \Delta), \ldots, c(k, \Delta)]$, where $c(i, \Delta)$ is the content of the block which tape i is scanning during segment Δ. Let also $f(\Delta)$ be the initial contents of those blocks which are visited by the DTM for the first time during the time segment Δ.
2. We use $q = [q(1), \ldots, q(t(n)^{\frac{1}{3}})]$, where $q(\Delta)$ is the state after time segment Δ. The result of the computation is determined by the state **after the final segment**.
3. How can we verify a guessed sequence of head positions and states?

Proof (6). Step 2: Relating the game with the problem (4).

1. DTM is deterministic and respects blocks: (1) $q(\Delta)$, $h(\Delta)$, and $c(\Delta)$ can be **uniquely determined from** $q(\Delta-1)$, $h(\Delta-1)$, and $c(\Delta_1),\ldots,c(\Delta_k)$ and $f(\Delta)$ by simulating time segment Δ. **This requires space $\mathbf{O}(kt(n)^{\frac{2}{3}})$.** (2) **Storing $c(\Delta)$ also requires space $\mathbf{O}(kt(n)^{\frac{2}{3}})$.**
2. $f(\Delta)$, $q(\Delta-1)$, and $h(\Delta-1)$ can be determined from the guessed sequences q and h. The edges into vertex $v(\Delta)$ in the graph G_k give us the vertices associated with $c(\Delta_1),\ldots,c(\Delta_k)$. How do we simulate the DTM? We first compute $c(1)$, then $c(2)$, etc.

Proof (7). Step 2: Relating the game with the problem (5).

1. For the simulation we do not need to store the contents of the blocks that correspond to all vertices (that would be too much): we can always reconstruct them. What is the minimal number of blocks that must be stored (at any time) to do the simulation?
2. This is exactly the pebble game.

Proof (8). Step 3: Intermediate result.

We aim to show

$$\mathbf{DTIME}(t(n)) \subseteq \mathbf{NSPACE}(\frac{t(n)}{\lg t(n)})$$

How can we build a NTM M' that simulates our $t(n) \lg t(n)$ time bounded k-DTM M in space $t(n)$?

Using the results above, we may assume that M **respects blocks**. We now guess a sequence of states q' and a sequence of head positions h'. These sequences have length at most $t(n)^{\frac{1}{3}}$ and it takes space $t(n)^{\frac{1}{3}}$ to write down q' and space $t(n)^{\frac{1}{3}} \lg t(n)$ to write down h'.

Proof (9). Step 3: Intermediate result (2).

From h' we construct the graph G from Slide 496. This graph has $t(n)^{\frac{1}{3}}$ many nodes and can be written down in space $t(n)^{\frac{1}{3}} \lg t(n)$.
Lemma 3.23 ensures that there is a strategy to move a pebble to the output node **by never using more than $t(n)^{\frac{1}{3}} \lg t(n)$ pebbles**. For this we need at most $2^{t(n)^{\frac{1}{3}}}$ many moves, because this is also the number of patterns (and repeating a pattern does not make sense).
The simulation is given on the next slide.

Proof (10). Step 3: Intermediate result (3).

1. Guess q' and h'.
2. For $1 \leqslant i \leqslant 2^{t(n)^{\frac{1}{3}}}$ do
   1. Guess move i of the above strategy.
   2. If a pebble is placed on $v(\Delta)$, then
      1. Compute and store $q(\Delta)$, $h(\Delta)$, and $c(\Delta)$.
      2. If $q(\Delta) \neq q'(\Delta)$ or $h(\Delta) \neq h'(\Delta)$ then reject.
      3. If the space used is greater or equal to n, then reject.
   3. If a pebble is removed from $v(\Delta)$ then erase $q(\Delta)$, $h(\Delta)$, and $c(\Delta)$ from the working tape.
3. If the simulation is **not complete**, then reject.

Only those $c(\Delta)$ are stored (after move m), for which there is a pebble on the corresponding vertex after this move. Storing $c(\Delta)$ requires space $\mathbf{O}(kt(n)^{\frac{2}{3}})$, multiplied with $t(n)^{\frac{1}{3}} \lg t(n)$ **proves the intermediate result**.

Proof (11). Step 4: Making the simulation deterministic.

We have two nondeterministic steps in our simulation.

Guessing q' and h': We cycle through all sequences.

Guessing the pebble move: Construct NTM: Given G, a pattern D of pebbles on G, and a number m between 1 and $2^{t(n)^{\frac{1}{3}}}$, it prints out the first move of a strategy that starts with D and moves the pebble on the output node of G (never using more than $\mathbf{O}(t(n)^{\frac{1}{3}} \lg t(n))$ pebbles and making at most $2^{t(n)^{\frac{1}{3}}} \lg t(n) - m$ moves, **provided such a strategy exists**. This NTM needs nd. space $\mathbf{O}(t(n)^{\frac{1}{3}} \lg t(n))$. By Theorem 3.21 (Savitch) there is a DTM using $\mathbf{O}(t(n)^{\frac{2}{3}} \lg^2 t(n))$ det. space. So the overall machine is below the space bound and the next move can be determined deterministically.

Proof (12). Step 5: Putting it all together.

1. We have shown that for space constructible functions $t(n)$: $\mathbf{DTIME}(t(n)) \subseteq \mathbf{DSPACE}(\frac{t(n)}{\lg t(n)})$.
2. The assumption **space constructible** can be lifted: We successively simulate the proof of the Theorem for $t(n) = 1, 2, \ldots$ until the simulation gets through.
3. We note without proof that the following holds. If $t(n)$ and $\delta(n)$ are both space constructible and $\lim_{n\to\infty} \delta(n) = \infty$,

$$\mathbf{DTIME}(t(n)) \subsetneq \mathbf{DSPACE}(t(n)) \cap \mathbf{DTIME}(\delta(n)t(n) \lg t(n))$$

What about

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{NSPACE}\left(\frac{t(n)}{\lg t(n)}\right)?$$

Might be true but up to now (2017) it has not been proved.

Here is one of the very few results of strict containment (without proof).

Theorem 3.24 (Linear det. $\neq$ linear nondet.)

We have the following strict containment *between the* deterministic *and the* nondeterministic linear time *complexity classes:*

$$\mathbf{DTIME}(n) \subsetneqq \mathbf{NTIME}(n)$$

3.4 More Properties

Theorem 3.25 (Union Theorem for SPACE)

Let $f_i(n)$, $i = 1, 2, \ldots$ be an enumeration of recursive functions with $f_i(n) \lneqq f_{i+1}(n)$ for all $n \in \mathbb{N}$. Then there **exists a recursive function $S(n)$** *with*

$$\mathbf{DSPACE}(S(n)) = \bigcup_{i \geqslant 1} \mathbf{DSPACE}(f_i(n)).$$

Proof of Theorem 3.25

Construct a function $\mathbf{S}(n)$ satisfying

1. For all i: $\mathbf{S}(n) \geqslant f_i(n)$ **almost everywhere (a.e)**.
2. If $S_j(n)$ is the space complexity of a DTM M_j and for each i: $S_j(n) \geqslant f_i(n)$ for **infinitely many** n's, then $S_j(n_0) \ngeqq \mathbf{S}(n_0)$ for some n_0.

Note that a language is accepted by a TM that is $S(n)$ space bounded

iff it is accepted by a TM that is $S(n)$ space bounded **a.e**.

Therefore (1) proves "$\supseteq$", (2) proves "$\subseteq$".

$\leadsto$ **blackboard 3.6**

Proof of Theorem 3.25 (2)

Note that setting $S(n) = f_n(n)$ leads to a class that is **too big** (condition (2) is not satisfied). **How can we modify $S(n)$ so that it is less than $S_j(n_0)$ for some n_0 (which might be different for each i)?**

We guess for each M_j a i_j s.t. $f_{i_j}(n) \geqslant S_j(n)$ **a.e.** This guess can be made deterministic: if it was wrong, we guess a larger value and for some n' we define $S(n')$ to be less than $S_j(n')$.

- Case 1: S_j grows faster than any f_i: **S** is infinitely often less than S_j.
- Case 2: Some f_i is almost everywhere greater than S_j: Eventually we guess such an f_i and stop assigning values of **S** less than S_j.

Proof of Theorem 3.25 (3)

List	$=$	$\emptyset$
for	$n =$	$1, 2, \ldots$ **do**
	if	for all $i_j = k$ on *List*, $f_k(n) \geqslant S_j(n)$
	then	add $i_n = n$ to *List* and set $\mathbf{S}(n) = f_n(n)$
	else	

begin
Among all guesses on *List* with $f_k(n) \lneqq S_j(n)$,
let $i_j = k$ be the guess with the smallest k and the smallest j (for that k).
Let $\mathbf{S}(n) = f_k(n)$.
Replace $i_j = k$ by $i_j = n$ on *List*.
Add $i_n = n$ to *List*.
end

This defines exactly the $\mathbf{S}$ we need.

Theorem 3.26 (Union Theorem for TIME)

Let $f_i(n)$, $i = 1, 2, \dots$ *be an enumeration of recursive functions with* $f_i(n) \lneqq f_{i+1}(n)$ *for all* $n \in \mathbb{N}$. *Then there* exists a recursive function $\mathbf{S}(n)$ *with*

$$\mathbf{DTIME}(\mathbf{S}(n)) = \bigcup_{i \geqslant 1} \mathbf{DTIME}(f_i(n)).$$

- For particular functions (**fully space/time constructible**), time and space hierarchies are dense.
- This is not true for all functions: there can be arbitrarily large gaps.

Theorem 3.27 (Borodin's Gap Theorem)

Let $g(n)$ be a total recursive function with $g(n) \geqq n$. Then there exists a total recursive function $\mathbf{S}(n)$ such that

$$\mathbf{DSPACE}(\mathbf{S}(n)) = \mathbf{DSPACE}(g(\mathbf{S}(n))).$$

Proof.

For an enumeration $M_1, \ldots, M_i, \ldots$ of DTM's, we denote by $S_i(n)$ the maximum number of cells used by M_i on an input of length n (why is this a **partial** recursive function?). We define $\mathbf{S} : \mathbb{N} \to \mathbb{N}$ as follows: for all i

1. $\mathbf{S}(n) \geqslant S_i(n)$ **a. e.**,
2. $S_i(n) \geqslant g(\mathbf{S}(n))$ **infinitely often**.

Assume now the two complexity classes are not identical, i.e. there is a language L in $\mathbf{DSPACE}(g(\mathbf{S}(n)))$ but not in $\mathbf{DSPACE}(\mathbf{S}(n))$. Then $L = L(M_i)$ for some i with $S_i(n) \leqslant g(\mathbf{S}(n))$ for all n. Together with the first condition we have $S_i(n) \leqslant \mathbf{S}(n)$ **a. e.**, which means L is in $\mathbf{DSPACE}(\mathbf{S}(n))$. □

Proof of Theorem 3.27 (2)

How can we define the function S with the above properties?

Given n, we cannot simply compute the largest finite value of $S_i(n)$ (for $1 \leqslant i \leqslant n$) because the $S_i(n)$ might be undefined.

Consider the following procedure to define S:

$j = 1$
while $\exists M_i \ (1 \leqslant i \leqslant n) \quad s.t.\ j+1 \leqslant S_i(n) \leqslant g(j)$
do $j = S_i(n)$
$S(n) = j$

This is a **total recursive** function as can be seen by counting ID's.

While we have proved Borodin's gap theorem only for DSPACE, its **analogues for NSPACE, DTIME and NTIME are true as well**.

Corollary 3.28 (Unintuitive Consequences)

There exist total recursive functions f such that

$$\mathbf{DSPACE}(f(n)) = \mathbf{DTIME}(f(n)) = \mathbf{NSPACE}(f(n)) = \mathbf{NTIME}(f(n))$$

$\rightsquigarrow$ **blackboard 3.7**

- Any DTM can be sped up by a linear factor (time and space): see Theorem 3.4 and Theorem 3.6.
- But there are languages where no best program exists: they can always be sped up significantly (not just by a constant).

Theorem 3.29 (Blum's Speed-Up Theorem)

For each total recursive function $\mathbf{r}(n)$*, there is a recursive language* L *such that the following holds.*
For each DTM M_i *accepting* L *in space* $S_i(n)$*, there is another DTM* M_j *accepting* L *in space* $S_j(n)$ *with:*

$$\mathbf{r}(S_j(n)) \leqslant S_i(n) \ \mathbf{a.e.}$$

Proof.

Wlog we assume $r(n)$ is monotonically nondecreasing, fully space-constructible and $r(n) \geqslant n^2$. We define the fully space-constructible function h

$$h(1) = 2, h(n) = \mathbf{r}(h(n-1))$$

Let $M_1, \ldots, M_i, \ldots$ be an enumeration of **offline** DTM's. We construct L such that

1. If $L(M_i) = L$, then $S_i(n) \geqslant h(n-1)$ **a.e.**,
2. for each k there is a DTM M_j with $L(M_j) = L$ and $S_j(n) \leqslant h(n-k)$.

These conditions imply (choose j s.t. $S_j(n) \leqslant h(n-i-1)$)

for each M_i accepting L there is M_j accepting L with:
$S_i(n) \geqslant \mathbf{r}(S_j(n))$ **a.e.**
$\rightsquigarrow$ ***blackboard 3.8***

4. EXPSPACE

4 **EXPSPACE**

- Complexity classes
- Reductions
- Structure of NP
- Oracles for P, NP
- **PH**: Polynomial Hierarchy
- Structure of PSPACE
- **EXPTIME/EXPSPACE**

Motivation

We consider the structure of **EXPSPACE=NEXPSPACE** and its subclasses: EXP, PSPACE, NP, P, L and NL. We also investigate:

1. The notion of (1) polynomial many-one reduction (Karp), and (2) logarithmic space many-one reduction. A problem is reduced to a second one: It is **at most as difficult** as the second one.
2. The notion of a complete/hard problem in a complexity class: such a problem is among the most difficult in this class.
3. Can diagonalization settle the P =NP question?
4. The polynomial hierarchy PH. Alternating TM's and Oracle TM's.
5. Beyond PSPACE: EXP, NEXP, EXPSPACE.

Brief Kurt Gödels an Johann von Neumann

Princeton, 20./III. 1956

Lieber Herr v. Neumann!

Ich habe mit größtem Bedauern von Ihrer Erkrankung gehört. Die Nachricht kam mir ganz unerwartet. Morgenstern hatte mir zwar schon im Sommer von einem Schwächeanfall erzählt, den Sie einmal hatten, aber er meinte damals, dass dem keine grössere Bedeutung beizumessen sei. Wie ich höre, haben Sie sich in den letzten Monaten einer radikalen Behandlung unterzogen u. ich freue mich, dass diese den gewünschten Erfolg hatte u. es Ihnen jetzt besser geht. Ich hoffe u. wünsche Ihnen, dass Ihr Zustand sich bald noch weiter bessert u. dass die neuesten Errungenschaften der Medizin, wenn möglich, zu einer vollständigen Heilung führen mögen.

Da Sie sich, wie ich höre, jetzt kräftiger fühlen, möchte ich mir erlauben, Ihnen über ein mathematisches Problem zu schreiben, über das mich Ihre Ansicht sehr interessieren würde: Man kann offenbar leicht eine Turingmaschine konstruieren, welche von jeder Formel F des engeren Funktionenkalküls u. jeder natürl. Zahl n zu entscheiden gestattet, ob F einen Beweis der Länge n hat [Länge = Anzahl der Symbole]. Sei $\Psi(F, n)$ die Anzahl der Schritte, die die Maschine dazu benötigt u. sei $\Phi(n) = \max_F \Psi(F, n)$. Die Frage ist, wie rasch $\Phi(n)$ für eine optimale Maschine wächst. Man kann zeigen $\Phi(n) \geqslant kn$. Wenn es wirklich eine Maschine mit $\Phi(n) \sim kn$ (oder auch nur $\sim kn^2$) gäbe, hätte das Folgerungen von der grössten Tragweite. Es würde nämlich offenbar bedeuten, dass man trotz der Unlösbarkeit des Entscheidungsproblems die Denkarbeit des Mathematikers bei ja-oder-nein Fragen vollständig durch Maschinen ersetzen könnte. Man müsste ja bloss das n so gross wählen, dass, wenn die Maschine kein Resultat liefert, es auch keinen Sinn hat über das Problem nachzudenken. Nun scheint es mir aber durchaus im Bereich der Möglichkeit zu liegen, dass $\Phi(n)$ so langsam wächst.

Denn 1.) scheint $\Phi(n) \geqslant k\Delta n$ die einzige Abschätzung zu sein, die man durch eine Verallgemeinerung des Beweises für die Unlösbarkeit des Entscheidungsproblems erhalten kann; 2.) bedeutet ja $\Phi(n) \sim kn$ (oder $\sim kn^2$) bloss, dass die Anzahl der Schritte gegenüber dem blossen Probieren von N auf $\log N$ (oder $(\log N)^2$) verringert werden kann. So starke Verringerungen kommen aber bei anderen finiten Problemen durchaus vor, z.B. bei der Berechnung eines quadratischen Restsymbols durch wiederholte Anwendung des Reziprozitätsgesetzes. Es wäre interessant zu wissen, wie es damit z.B. bei der Feststellung, ob eine Zahl Primzahl ist, steht u. wie stark im allgemeinen bei finiten kombinatorischen Problemen die Anzahl der Schritte gegenüber dem blossen Probieren verringert werden kann.

Ich weiss nicht, ob Sie gehört haben, dass "Posts problem" (ob es unter den Problemen $(\exists y)\Phi(y, x)$ mit rekursivem Φ Grade der Unlösbarkeit gibt) von einem ganz jungen Mann namens Richard Friedberg in positivem Sinn gelöst wurde. Die Lösung, abgesehen von der Aufstellung der Axiome, ist sehr elegant. Leider will Friedberg nicht Mathematik, sondern Medizin studieren (scheinbar unter dem Einfluss seines Vaters).

Was halten Sie übrigens von den Bestrebungen, die Analysis auf die verzweigte Typentheorie zu begründen, die neuerdings wieder in Schwung gekommen sind? Es ist Ihnen wahrscheinlich bekannt, dass Paul Lorenzen dabei bis zur Theorie des Lebesgueschen Masses vorgedrungen ist. Aber ich glaube, dass in wichtigen Teilen der Analysis nicht eliminierbare imprädikative Schlussweisen vorkommen. Ich würde mich sehr freuen, von Ihnen persönlich etwas zu hören; u. bitte lassen Sie es mich wissen, wenn ich irgend etwas für Sie tun kann.

Mit besten Grüssen u. Wünschen, auch an Ihre Frau Gemahlin.
Ihr sehr ergebener

Kurt Gödel

PS. Ich gratuliere Ihnen bestens zu der Auszeichnung, die Ihnen von der amerik. Regierung verliehen wurde.

A useful blog: https://rjlipton.wordpress.com.

Lipton maintains a very interesting blog. There is a translation from Gödels lost letter, many more historical remarks as well as very recent developments, e.g. the quasipolynomial complexity of graph-isomorphism claimed by Bobai in November 2015).

Princeton, 20 March 1956
Dear Mr. von Neumann:
With the greatest sorrow I have learned of your illness. The news came to me as quite unexpected. Morgenstern already last summer told me of a bout of weakness you once had, but at that time he thought that this was not of any greater significance. As I hear, in the last months you have undergone a radical treatment and I am happy that this treatment was successful as desired, and that you are now doing better. I hope and wish for you that your condition will soon improve even more and that the newest medical discoveries, if possible, will lead to a complete recovery.
Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let Ψ(F,n) be the number of steps the machine requires for this and let $\Phi(n) = \max_F \Psi(F,n)$. The question is how fast Φ(n) grows for an optimal machine. One can show that $\Psi(n) \geqslant kn$. If there really were a machine with $\Phi(n) \sim k\,n$ (or even $\sim k\,n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that Φ(n) grows that slowly. Since it seems that $\Phi(n) \geqslant k\,n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $\Phi(n) \sim kn$ (or $\sim kn^2$) only means that the number of steps as opposed to trial and error can be reduced from N to log N (or $(\log N)^2$). However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.
I do not know if you have heard that “Post’s problem”, whether there are degrees of unsolvability among problems of the form ($\exists$ y) Φ(y,x), where Φ is recursive, has been solved in the positive sense by a very young man by the name of Richard Friedberg. The solution is very elegant. Unfortunately, Friedberg does not intend to study mathematics, but rather medicine (apparently under the influence of his father). By the way, what do you think of the attempts to build the foundations of analysis on ramified type theory, which have recently gained momentum? You are probably aware that Paul Lorenzen has pushed ahead with this approach to the theory of Lebesgue measure. However, I believe that in important parts of analysis non-eliminable impredicative proof methods do appear.
I would be very happy to hear something from you personally. Please let me know if there is something that I can do for you. With my best greetings and wishes, as well to your wife,
Sincerely yours,
Kurt Gödel
P.S. I heartily congratulate you on the award that the American government has given to you.

4.1 Complexity classes

Here are some classical complexity classes:

Definition 4.1 (From L to NEXPSPACE)

$$
\begin{array}{lcl}
\mathbf{L} & := & \mathbf{DSPACE}(\log n) \\
\mathbf{NL} & := & \mathbf{NSPACE}(\log n) \\
\mathbf{P} & := & \bigcup_{i\geqslant 1} \mathbf{DTIME}(n^i) \\
\mathbf{NP} & := & \bigcup_{i\geqslant 1} \mathbf{NTIME}(n^i) \\
\mathbf{PSPACE} & := & \bigcup_{i\geqslant 1} \mathbf{DSPACE}(n^i) \\
\mathbf{NSPACE} & := & \bigcup_{i\geqslant 1} \mathbf{NSPACE}(n^i) \\
\mathbf{EXP} & := & \bigcup_{i\geqslant 1} \mathbf{DTIME}(2^{n^i}) \\
\mathbf{NEXP} & := & \bigcup_{i\geqslant 1} \mathbf{NTIME}(2^{n^i}) \\
\mathbf{EXPSPACE} & := & \bigcup_{i\geqslant 1} \mathbf{DSPACE}(2^{n^i}) \\
\mathbf{NEXPSPACE} & := & \bigcup_{i\geqslant 1} \mathbf{NSPACE}(2^{n^i})
\end{array}
$$

The unions are complexity classes

Note that it is not clear that the unions on the preceding page are indeed complexity classes, i.e. they have the form $\mathbf{C}(f(n))$ for a function $f(n)$. We need Theorem 3.25 on Slide 508 to establish this fact.

Definition 4.2 ($\mathbf{co\text{-}C}$)

Let $\mathbf{C}$ be any of the above complexity classes. We define $\mathbf{co\text{-}C}$ as the class of languages L, where the complement $\overline{L}$ ($= \Sigma^* \backslash L$) is in $\mathbf{C}$.

Obviously, all classes $\mathbf{C}$ defined on deterministic TM's are closed under complements. For those classes, $\mathbf{co\text{-}C} = \mathbf{C}$.

Nondeterministic classes

If $\mathbf{C}(f(n)) \subseteq \mathbf{co\text{-}C}(f(n))$ then $\mathbf{C}(f(n)) = \mathbf{co\text{-}C}(f(n))$, because $\overline{\overline{L}} = L$.

We define a co-C machine M. This is a machine M which passes its input to a machine M' running in **C**, waits for the answer and switches the result: reject becomes accept and vice versa.

- This is uncritical, if the machines considered are deterministic: M accepts an input w **iff** M' does not accept it.
- If **C** is any non-deterministic class, which means that M' is a NTM, then there might be accepting and non-accepting computations.
- Remember that a NTM does not accept an input w, if all computations end in non-accepting states. We also say that the machine rejects the input.
- We define: M accepts an input **iff** M' rejects it.

  M does not accept w **iff** M' accepts w (i.e. if there is at least one accepting computation).

For all nondeterministic classes, with the exception of

- **NEXPSPACE**, which is identical to **EXPSPACE** = **co-EXPSPACE**,
- **NSPACE**, which is identical to **PSPACE** = **co-PSPACE**,
- **NL**, where **NL** = **co-NL** follows from the famous result of Immerman/Szelepcsényi (Theorem 4.4, Slide 533, see also Theorem 1.72 on Slide 161),

it is unknown whether they are closed under complements.

Question:

What are the relations between the classes from Definition 4.1?

With Savitch's Theorem 3.21 (Slide 487), we get (using the hierarchy on Slide 476, Theorem 3.18):

$$\begin{array}{rcl} \mathbf{PSPACE} & = & \mathbf{NSPACE} \\ \bigcup_{i \geqslant 1} \mathbf{DSPACE}(n^i) & = & \bigcup_{i \geqslant 1} \mathbf{NSPACE}(n^i) \end{array}$$

Similarly, we get

$$\mathbf{EXPSPACE} = \mathbf{NEXPSPACE}$$

- Using Theorem 3.20 (3), we have $\mathbf{NL} \subseteq \mathbf{P}$.
- We get (Theorem 3.17):

$$\mathbf{L} \subseteqq \mathbf{NL} \subseteqq \mathbf{P} \subseteqq \mathbf{NP} \subseteqq \mathbf{PSPACE}$$

where at least one inclusion is strict (**which one is unknown**): the hierarchy $\mathbf{DSPACE}(\log^i n)$ is strict (see Slide 476).

Similarly:

$$\mathbf{P} \subsetneqq \mathbf{EXP} \subsetneqq \mathbf{EXPSPACE}$$

(because the hierarchies $\mathbf{DTIME}(n^i)$ and $\mathbf{DTIME}(2^{n^i})$ are strict), and

$$\mathbf{PSPACE} \subsetneqq \mathbf{EXPSPACE}$$

(because the hierarchy $\mathbf{DSPACE}(n^i)$ is strict).

Obviously

$$\mathbf{EXP} \subseteq \mathbf{NEXP} \subseteq \mathbf{EXPSPACE}.$$

With Theorem 3.20 (2), we have

$$\mathbf{PSPACE} \subseteq \mathbf{EXP}.$$

Lemma 4.3 ($\mathbf{NSPACE}(n)$ =Type-1)

The class of contextsensitive *languages is exactly the class* $\mathbf{NSPACE}(n)$.

Proof.

The class CSL is defined as a particular class of grammars. By Theorem 1.63 we get the connection to $\mathbf{NSPACE}(n)$. □

Theorem 4.4 (Immerman/Szelepcsényi)

Let $S(n) \geqslant \log n$. *Then*
$\mathbf{NSPACE}(S(n)) = \mathbf{co\text{-}NSPACE}(S(n))$.

Corollary 4.5 (CSL is closed under complements)

The class of contextsensitive languages is closed under complements.

Corollary 4.6 (Tight Space Hierarchy for $\mathrm{NSPACE}(S(n))$)

Let $S_2(n)$ be space constructible. We also assume, that both $S_1(n)$ and $S_2(n)$ are $\geqslant \log n$.

- *If* $\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$ *then* $\mathbf{NSPACE}(S_1(n)) \subsetneq \mathbf{NSPACE}(S_2(n))$.

Therefore we have an analogue of Theorem 3.17 for $\mathbf{NSPACE}(f(n))$.

(Proof of Corollary 4.6).

The proof follows the analogous statement for $\mathbf{DSPACE}(S(n))$ (see Theorem 3.17). There we used the fact that a **DTM can easily switch between acceptance and rejection**. This is not true for a NTM. Therefore we reword our definition on the top of Slide 475 as follows:

M **accepts** w <u>**iff**</u> (1) M can do the simulation in time $S_2(n)$ and (2) M_w terminates and accepts w (for a contradiction, we will use the complement of the language instead).

Obviously, $L(\mathrm{M})$ is in $\mathbf{NSPACE}(S_2(n))$.

Assume L is accepted by M in $\mathbf{NSPACE}(S_1(n))$. So there is $\overline{\mathrm{M}}$ accepting $\overline{L}$ (Theorem 4.4). Choose w where $\mathrm{M}_w = \overline{\mathrm{M}}$ and $\frac{S_1(|w|)}{S_2(|w|)}$ is small enough. Then M accepts w <u>**iff**</u> $\mathrm{M}_w = \overline{\mathrm{M}}$ does. This is a contradiction.

(Proof of Theorem 4.4).

The original formulation required that $S(n)$ be space constructible. But this is not needed. We base the proof on two lemmas, that are of independent interest and are somehow related to the proof of Theorem 4.15.

Lemma 4.7 (Number of Reachable Nodes)

Given a directed graph with n nodes and a particular node x, the number k *of* **nodes reachable from** x *can be computed in* $\mathbf{NSPACE}(\log n)$.

(Proof of Theorem 4.4 (2)).

The key in the proof of the theorem is the following lemma. This is exactly the concept of a **co-C** machine as introduced on Slide 526.

Lemma 4.8 (Rejecting NTM)

Given a directed graph, two nodes x, y of the graph and the number k of nodes reachable from x, there is NTM working in $\log n$ space for which the following holds:

If y is reachable from x, then all computations end in non-accepting states;

if y is not reachable from x, then there is at least one accepting computation.

(Proof of Theorem 4.4 (3)).

Let L be a language in $\mathbf{NSPACE}(S(n))$. So there is a NTM machine M, space bounded by $(S(n))$, accepting L. We have to construct a machine M', space bounded by $S(n)$, accepting $\overline{L}$. Given input w, we consider the following **directed graph**

nodes: all configurations of M.

edges: there is an edge from $config$ to $config'$, if M can reach $config'$ from $config$ in one step.

Wlog we assume there is only one accepting configuration (we can assume that the NTM has only one accept state). Note that $w \in \overline{L}$ **iff** there is no computation ending in the accept state. We define the NTM M' as follows:

1. Compute the number of nodes reachable from $config_{initial}$.
2. Use this number and check, whether $config_{accept}$ is reachable from $config_{initial}$. In that case reject, otherwise accept.

(Proof of Theorem 4.4 (4)).

Note that we do not explicitly construct (and write down) the graph: it is implicitly represented by M.

Obviously the following holds:

- M' accepts $\overline{L}$.
- M' is $\log n$ space-bounded in the number n of nodes in the graph (the two lemmas).

How many nodes are there?
$\mathbf{O}(c^{S(n)})$, where c is the size of the alphabet.
Thus M' is $S(n)$ space-bounded.

Therefore $L \in \mathbf{co\text{-}NSPACE}(S(n))$, thus:
$\mathbf{NSPACE}(S(n)) \subseteq \mathbf{co\text{-}NSPACE}(S(n))$. The converse implication follows from that immediately (see the remark on Slide 525).

(Proof of Lemma 4.7 (1)).

Let S_i be the set of nodes **reachable from x in at most i steps**. We compute S_k from S_{k-1}.
The problem in doing this is to determine whether a node is reachable in one step from S_k by only knowing $|S_{k-1}|$: and doing it in $\mathbf{NSPACE}(\log n)$.
The algorithm to achieve this follows on Slide 542.

- The outer loop (for **k**) first gets $|S_0| = |\{x\}| = 1$. The next iteration, $k = 1$, determines $|S_1|$, the neighbours of x. In each iteration, the nodes from the previous one, S_{k-1}, are counted again.
- The l-counter is incremented (or not) for each w (if the flag is true) and thus counts S_k.
- For fixed w, the counter i counts the number of nodes in S_i that have been tried.

(Proof of Lemma 4.7 (2)).

- The inner **for**-loop (lines 6–15) guesses nodes v in S_i. When one of them is adjacent to w, the flag is set to true. If there is none, i is incremented and the search goes on. So eventually an i, small as possible, is computed. This i is compared to $|S_{k-1}|$: if it is smaller then **not all possible candidates have been tried**. This is because the **nondeterministic guesses** were wrong and some v's were discarded. In that case, one has to start all over again (reject).
- At the end of the **for**-loop, the flag tells us whether $w \in S_{i+1}$.
- l is only incremented, when all nodes in S_i have been checked.

```
1:  for k=0, ..., n-1 do
2:      l := 0
3:      for all nodes w of G do
4:          i := 0
5:          flag := false
6:          for all nodes v of G do
7:              nondeterministically goto line 8 or line 14:
8:              if v is reachable from x in at most i steps then
9:                  if w is a neighbour of v then
10:                     flag := true
11:                 else
12:                     i := i + 1  % only when v ∈ S_i and w not adjacent to v
13:                 end if
14:             end if
15:         end for  % if flag is true, then w ∈ S_{i+1}
16:         if i ≨ |S_{k-1}| then
17:             reject  % because not all candidates have been checked (line 7)
18:         else
19:             if flag = true then
20:                 l := l + 1  % the w is counted as an element of S_k
21:             end if
22:         end if
23:     end for
24: end for
25: |S_k| := l
```

(Proof of Lemma 4.8).

It is easy to define a machine which **accepts** if the two nodes are reachable: see the proof of Theorem 4.15. **Here we do not even need the number k.**

But we want a NTM that **rejects**. The idea is the following

- If there are k different nodes reachable from x and they are all different from y, then **y is not reachable from x**.

The algorithm to achieve this follows on Slide 544. □

```
1:  i := 0
2:  for all nodes v of G do
3:      nondeterministically goto line 4: or line 11:
4:      if v is reachable from x then
5:          if v = x then
6:              reject
7:          else
8:              i := i + 1
9:          end if
10:     end if
11: end for
12: if i ≨ k then
13:     reject
14: else
15:     accept
16: end if
```

4.2 Reductions

We have already introduced in Chapter 2 several notions of reduction. We have also defined the most difficult problems in a given class: those, which all others in the class can be reduced to.

We need fine-grained reductions.

The notion of a most difficult problem depends on the underlying notion of reduction.

Such problems are called complete in the given class wrt the notion of reducibility.

Definition 4.9 (P-Time-, $\log$-Space Reducibility)

Let L_1, L_2 be two languages.

Polynomial Time: L_2 is P-time reducible to L_1 if there is a polynomial time-bounded DTM M such that for each input w a $\mathrm{M}(w)$ is generated, such that

$$w \in L_2 \text{ \underline{iff} } \mathrm{M}(w) \in L_1 \quad (L_2 \preceq_{pol} L_1)$$

Logarithmic Space: L_2 is $\log$-space reducible to L_1 if there is a $\log$ space-bounded offline-DTM M, such that for each input w an output $\mathrm{M}(w)$ is generated, such that

$$w \in L_2 \text{ \underline{iff} } \mathrm{M}(w) \in L_1 \quad (L_2 \preceq_{log} L_1)$$

The output tape is “read-only” and the head never moves to the left.

Reductions

How does this notion relate to the ones defined in Definition 2.85 (Slide 423)?

Lemma 4.10 (P-Time- and log Space Reductions)

1. *Let* $L_2 \preceq_{pol} L_1$. *Then:* $L_2 \in \mathbf{NP}$ *if* $L_1 \in \mathbf{NP}$,
   $L_2 \in \mathbf{P}$ *if* $L_1 \in \mathbf{P}$.
2. *Let* $L_2 \preceq_{log} L_1$. *Then*
   $L_2 \in \mathbf{P}$ *if* $L_1 \in \mathbf{P}$
   $L_2 \in \mathbf{DSPACE}(\log^k n)$ *if* $L_1 \in \mathbf{DSPACE}(\log^k n)$
   $L_2 \in \mathbf{NSPACE}(\log^k n)$ *if* $L_1 \in \mathbf{NSPACE}(\log^k n)$
3. *The composition of two* log *space reductions is again a* log*-space reduction. Similarly for polynomial time reductions.*
4. log*-space reducibility implies polynomial time reducibility.*

Definition 4.11 (Complete, Hard)

Let $\mathbf{C}$ be any complexity class. For classes below or equal to $\mathbf{NP}$, we define:

- A language L is called $\mathbf{C}$-complete if $L \in \mathbf{C}$ and each language $L' \in \mathbf{C}$ is $\log$-space reducible to L.
- A language L is called $\mathbf{C}$-hard if each language $L' \in \mathbf{C}$ is $\log$-space reducible to L.

For classes above or equal to $\mathbf{PSPACE}$, we define:

- A language L is called $\mathbf{C}$-complete if $L \in \mathbf{C}$ and each language $L' \in \mathbf{C}$ is $\mathbf{P}$-time reducible to L.
- A language L is called $\mathbf{C}$-hard if each language $L' \in \mathbf{C}$ is $\mathbf{P}$-time reducible to L.

Attention:

- If there exists one $\mathbf{NP}$-hard problem in $\mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
- If there exists one $\mathbf{PSPACE}$-hard problem is in $\mathbf{P}$, then $\mathbf{P} = \mathbf{PSPACE}$.
- If $\mathbf{P} \neq \mathbf{NP}$, then no $\mathbf{NP}$-complete problem is solvable in polynomial time.
- All problems in $\mathbf{L}$ are $\mathbf{L}$-complete (wrt. $\leq_{\log}$).

4.3 Structure of $\mathbf{NP}$

Intuitively:

- Problems in P can be efficiently solved. Those in NP need most probably exponential time.
- NP is the class of statements with easily verifiable proofs.
- PSPACE is a huge class, much bigger than P or NP.
- **L**=$\mathrm{DSPACE}(\log n)$ is a small class in P and even smaller in PSPACE.
- **NL**=$\mathrm{NSPACE}(\log n)$ is also small in P and even smaller in PSPACE.

Search Problems

On the next few slides we introduce the $\mathbf{P} = \mathbf{NP}$ problem in terms of search problems.

- A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is called polynomially-bounded if there is a polynomial p such that for all $\langle x, y \rangle \in R$: $|y| \leqslant p(|x|)$.
- Given a polynomially-bounded binary relation R, the associated search problem is the following: Given x find y such that $\langle x, y \rangle \in R$ or show that such y does not exist.
- $\mathbf{P}^{search}$ consists of those relations, that can be computed in polynomial time.
- $\mathbf{NP}^{search}$ consists of those relations, where a potential candidate $\langle x, y \rangle$ can be verified in polynomial time.

Definition 4.12 ($\mathbf{P}^{search}$, $\mathbf{NP}^{search}$)

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a polynomially bounded binary relation.

$\mathbf{P}^{search}$: is the class of associated search problems that are solvable in polynomial time: There is a polynomial time algorithm that given x computes y such that $\langle x, y \rangle \in R$ if it exists, or outputs "No" otherwise.

$\mathbf{NP}^{search}$: is the class of associated search problems that are verifiable in polynomial time: There is a polynomial time algorithm that given $\langle x, y \rangle$ determines whether $\langle x, y \rangle \in R$ or not.

Theorem 4.13

$\mathrm{P} = \mathrm{NP}$ ***iff*** $\mathrm{P}^{search} = \mathrm{NP}^{search}$

Proof.

$\rightsquigarrow$ **exercise**

We define:

Definition 4.14 (Reachability: STCON, USTCON)

How to decide whether there is a path between two vertices in a graph. This problem can be stated for directed and for undirected graphs.

$$L_{\text{STCON}} := \{\langle G, s, t\rangle : \; G \text{ directed graph, } s, t \text{ vertices there is a directed path from } s \text{ to } t\}$$

$$L_{\text{USTCON}} := \{\langle G, s, t\rangle : \; G \text{ a graph, } s, t \text{ vertices there is a path between } s \text{ to } t\}$$

While STCON is complete for $\mathbf{NL}$ (relatively easy), USTCON is even in $\mathbf{L}$ (and thus complete for this class). The latter is a highly nontrivial result and was open for several decades.

Theorem 4.15 (Completeness of STCON and USTCON)

While USTCON (see Definition 4.14) is complete *for* $\mathbf{L} = \mathbf{DSPACE}(\log n)$*, STCON is* complete *for* $\mathbf{NL} = \mathbf{NSPACE}(\log n)$ *(both under* $\log$ *space reductions).*

(Proof of Theorem 4.15).

We only do STCON, the other result is too difficult for this course (note that USTCON $\in \mathbf{L}$ is the difficult part). Given a directed graph with n nodes and two nodes s, t.

- Start with s and nondeterministically guess the next node (it is also allowed not to move at all in this step).
- Keep only the current node on the tape, not the whole path, and the counter. Both need only $\log n$ space.
- Terminate after exactly $n - 1$ steps.
- Check, whether t is the current node.
- If yes: accept, else reject.

This shows that STCON is in $\mathbf{NL}$. □

(Proof of Theorem 4.15 (2)).

We have to show completeness of STCON under $\log$ space reductions. Let L be any language in $\mathbf{NSPACE}(\log n)$. Thus there is a NTM M accepting L and working in $\log$ space. We consider the ID's (instantaneous descriptions) of M when started with an input x.

- The length of an ID is $4 \log n$: the storage tape, the current state, the position of the input head and the position of the storage head.
- Note that we do not consider the input x, only the position of the input head (otherwise it would not be in $\log n$ space).

(Proof of Theorem 4.15 (3)).

- We describe a $\log n$ **space-bounded offline DTM** M', which works on an input x and produces a **directed graph** G_x with nodes s, t such that

$$\mathrm{M}(x)\downarrow \quad \underline{\textbf{iff}} \quad s \text{ is reachable from } t \text{ in } G_x$$

  - Nodes of G_x are the ID's of the previous slides
  - It starts with the initial ID (M started on x) and cycles through the ID's by simulating M.
  - But how does it know which ID next: x is not part of an ID?
  - It does so by positioning its head on the right place of its input x (which is **encoded** in the current ID).
  - It is obvious how and when to generate the edges from one ID to all possible successor ID's (note M is a NTM).
- The right hand side is an instance of STCON. Therefore we have reduced L (which was any language in $\mathbf{NSPACE}(\log n)$) to STCON.

To motivate the above intuitions we consider the problem to decide whether a given formula is satisfiable (satisfiability).

Definition 4.16 (Boolean Expressions)

Let $\Sigma_{\mathsf{sat}} := \{\wedge, \vee, \neg, (,), x, 0, 1\}$. This alphabet suffices, to define arbitrary boolean expressions w over the infinite variable set $\{x_1, x_2, \ldots\}$. A variable x_i is encoded by the sign x followed by i in binary notation.
As usual each boolean expression can be assigned a truthvalue **t** (true) or **f** (false) depending on the valuation of the variables.

Here are a few examples:

1. $(x_1 \vee \neg x_2) \wedge \neg x_1$ evaluates to **t** for $x_1 = \mathbf{f}$ and $x_2 = \mathbf{f}$.
2. $(x_1 \vee \neg x_2) \wedge \neg x_1$ evaluates to **f** for $x_1 = \mathbf{t}$ (and the value of x_2 does not matter) or for $x_1 = \mathbf{f}$ and $x_2 = \mathbf{t}$.
3. $x_1 \vee \neg x_1$ always evaluates to **t**.

The satisfiability problem is the problem to decide whether a given boolean expression can evaluate to **t**, i.e. to find an appropriate valuation of the variables.

Definition 4.17 (Satisfiability: SAT)

$$L_{\mathsf{sat}} := \{w \in \Sigma_{sat}^* : \text{ } w \text{ is a well formed formula, there is a valuation for the } x_i, \text{ s.t. } w \text{ evaluates to true } \}$$

Often it is simpler to assume that a formula is in a certain normalform. In particular, formulas of the form

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

play an important role.

Definition 4.18 (SAT, k-CNF, k-DNF)

A literal l_i is a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals. We define

DNF: A formula is in **disjunctive normalform**, if it is of the form $(l_{11} \wedge \ldots \wedge l_{1n_1}) \vee \ldots \vee (l_{m1} \wedge \ldots \wedge l_{mn_m})$.

CNF: A formula is in **conjunctive normalform**, if it is of the form
$(l_{11} \vee \ldots \vee l_{1n_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mn_m})$.

Definition (Continued):

k-DNF: A formula is in *k-DNF* if it is in DNF and each disjunction contains exactly k literals.

k-CNF: A formula is in *k-CNF* if it is in CNF and each disjunction contains exactly k literals.

Note that we could also define these normalforms by defining at most k literals. The reason is that we can always do padding and fill the formulae by dummy variables. While the formulae get longer, they only get longer by a constant amount (k is fixed and the length of the formulae is not restricted).

We define:

Definition 4.19 (Satisfiability: k-SAT)

$$L_{\text{k-sat}} := \{w \in \Sigma_{sat}^* : \begin{array}{l} w \text{ is in } k\text{-CNF}, \\ w \text{ is a well formed formula}, \\ \text{there is a valuation for the } x_i, \\ \text{s.t. } w \text{ evaluates to true}\} \end{array}$$

Lemma 4.20 (DNF and 2-SAT)

DNF: *Satisfiability of formulae in DNF is in $\mathbf{DTIME}(n \log n)$.*

2-SAT: *$L_{\text{2-sat}}$ is $\mathbf{NL}$-complete.*

Proof.

The first part is trivial. For the second one, it suffices to show that $L_{\text{2-sat}}$ is in $\mathbf{NL}$. Consider a directed graph: The **nodes** are the literals (and their negations) occurring in the formula. For each clause $L_1 \vee L_2$ we add a directed edge from L_1 to $\neg L_2$ and from L_2 to $\neg L_1$ (in case L is already negated, i.e. of the form $\neg C$, then $\neg L$ is considered as C).

It remains to compute the strongly connected components of the graph. The formula is satisfiable <u>iff</u> no strongly connected component contains a literal and its negation.

In that case, we get a valid valuation by setting A to true, if $\neg A$ is not reachable from A. Otherwise it is set to false.

This shows that STCON $\log$ space reduces to $L_{\text{2-sat}}$. □

Question:

How to show that a given problem is in a certain complexity class?

Answer

Reduce it to a known one

The same with a *complete* problem.
But we need one to start with!

NL: In that case, we used STCON and showed directly, that all $\mathbf{NSPACE}(\log n)$ languages reduce to it.

$\mathbf{NP}$**:** Here Cook showed in 1971 how to simulate the computation of any NTM that is polynomially time-bounded, by SAT.

Theorem 4.21 (Characterization of $\mathbf{NP}$)

*A language L is in $\mathbf{NP}$ **iff** there is a language L' in $\mathbf{P}$ and a $k \geqslant 0$, s.t. for all $w \in \Sigma$:*

$$w \in L \quad \textbf{\textit{iff}} \quad \text{there is a } c : \langle w, c \rangle \in L' \text{ and } |c| < |w|^k.$$

*c is called **witness** or **certificate**) of w in L. A DTM accepting L' is called **verifier** of L.*

Proof.

⇝ **exercise**

Colloquial formulation:

A decision problem is in $\mathbf{NP}$ **iff** **each yes instance has a small certificate** (i.e. its length is polynomial in the input), which can be verified in polynomial time.

Theorem 4.22 (SAT is $\mathbf{NP}$-complete)

L_{sat} is $\mathbf{NP}$-complete.

Proof.

The original proof was given by Cook in 1971. That $L_{\text{sat}} \in \mathbf{NP}$ is simple: just guess a valuation. Verifying it is trivial (in linear time).
Now let L be any language in $\mathbf{NP}$. Then there is a NTM M accepting L and a polynomial $p(n)$ such that M is $p(n)$ time-bounded.

- In the following we describe a $\log$ space algorithm that, on input x produces a formula ϕ_x with

M accepts x **iff** ϕ_x is satisfiable

(Proof of Theorem 4.22 (2)).

1. We simulate computations of M.
2. Wlog each computation has exactly the form (where for each ID $|\beta_i| = p(n)$): $\#\beta_0\#\beta_1\#\ldots\#\beta_{p(n)}$.
3. Wlog we use new symbols that encode in each β_i: (1) the state, (2) the symbol currently scanned, (3) the number $m_{\beta_i} \in \mathbb{N}$ encoding which move is taken in ID i to get to β_{i+1}. Let **F** be the set of symbols encoding an **accepting** state.
4. For each symbol X occurring in the computation and each $0 \leqslant i \lneqq (p(n)+1)^2$ we use boolean variables $c_{i,X}$. The idea is that $c_{i,X}$ is true **iff** the i'th symbol in the computation is identical to X. E.g. $c_{0,\#}, c_{p(n)+1,\#}, \ldots, c_{p(n)(p(n)+1),\#}$ are all true.

(Proof of Theorem 4.22 (3)).

We construct E_x such that

E_x is true for a valuation val of the $c_{i,X}$

iff

$\{c_{i,X} : val(c_{i,X}) = \mathbf{true}\}$ corresponds to a valid computation of M.

Our aim is to ensure that E_x expresses the following:

(1) String: For each j with $c_{j,Y} \in \{c_{i,X} : val(c_{i,X}) = \mathbf{true}\}$ (Y arbitrary), there is exactly one Y_0 such that c_{j,Y_0} is true.

(2) Initial: β_0 is the initial ID for M with input x.

(3) Final: The last ID contains an accepting state.

(4) Move: Each ID follows from the previous one by the move that is indicated by m_{β_i}.

(Proof of Theorem 4.22 (4)).

E_x is the conjunction of the following

(1)

$$\bigwedge_i \left[\bigvee_X c_{i,X} \wedge \neg \left(\bigvee_{X \neq Y} (c_{i,X} \wedge c_{i,Y}) \right) \right]$$

(2) $\rightsquigarrow$ **blackboard 4.1**

(3)

$$\bigvee_{p(n)(p(n)+1) \lneqq i \lneqq (p(n)+1)^2} \left(\bigvee_{X \in \mathbf{F}} c_{i,X} \right)$$

(4) We can easily define a predicate $\mathbf{Q}(W,X,Y,Z)$ which says that "symbol Z could appear in position j of some ID, if W,X,Y are the symbols in positions $j-1, j, j+1$ of the previous ID". We can also arrange that $\mathbf{Q}(W,\#,X,\#)$ is always true.

$$\bigwedge_{p(n) \lneqq j \lneqq (p(n)+1)^2} \left(\bigvee_{\mathbf{Q}(W,X,Y,Z)} (c_{j-p(n)-2,W} \wedge c_{j-p(n)-1,X} \wedge c_{j-p(n),Y} \wedge c_{j,Z} \right)$$

Example 4.23 (More $\mathbf{NP}$-Complete Problems)

- Is a 3-SAT formula satisfiable?
- Is a directed graph hamiltonian? (hamiltonian circle)
- Is there a clique of size k in a graph? (maximum clique)
- Is a graph colorable with 3 colors? (3-colorability)
- Given a finite set of numbers and a number n. Is there a subset which sums up to n?
  (subset sum)

The following relations are unknown:

1. $\mathbf{P} \neq \mathbf{NP}$.
2. $\mathbf{NP} \neq \mathbf{co\text{-}NP}$.
3. $\mathbf{P} \neq \mathbf{PSPACE}$.
4. $\mathbf{NP} \neq \mathbf{PSPACE}$.
5. $\mathbf{L} \neq \mathbf{NL}$.
6. $\mathbf{L} \neq \mathbf{NP}$.

But the following holds:

1. *If* $\mathbf{P} = \mathbf{NP}$ *then* $\mathbf{EXP} = \mathbf{NEXP}$.
2. $\mathbf{L} \neq \mathbf{PSPACE}$.
3. $\mathbf{P} \neq \mathbf{EXP}$.

Lemma 4.24 (Ladner, 1975)

If $\mathrm{NP} \neq \mathrm{P}$, then there are problems in NP *that are* not NP-complete.

Proof.

$\leadsto$ **blackboard 4.2** □

Candidates for such problems are Graph-Isomorphism (are two given graphs isomorphic?), or Factorization.

Is the complement of Graph-Isomorphism in NP?

Note (added in proof): Graph-Isomorphism is quasipolynomial (Nov. 2015, Bolyai). Many people expect it to be in P (might take another decade to prove).

Lemma 4.25

If $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ *contains a* $\mathbf{NP}$*-hard language, then* $\mathbf{NP} = \mathbf{co\text{-}NP}$.

Proof.

$\rightsquigarrow$ **blackboard 4.3**

Lemma 4.26

If $\mathbf{NP} \neq \mathbf{co\text{-}NP}$*, then there are non-trivial languages in* $\mathbf{NP}$ *that can not be reduced to any set in* $\mathbf{co\text{-}NP}$.

Proof (of Lemma 4.26).

Let $L' \in \mathbf{co\text{-}NP}$. Let L be nontrivial such that $L \leq_{pol} L'$. Thus $\overline{L} \leq_{pol} \overline{L'}$ via a reduction f. But then $\overline{L} \in \mathbf{NP}$ (because $\{\langle x, y\rangle : \langle f(x), y\rangle \in R''\}$ is a $\mathbf{NP}$ relation: R'' corresponds to $\overline{L'}$)). Then $L \in \mathbf{co\text{-}NP}$. □

In fact, it is conjectured that

$$\mathbf{P} \subsetneq \mathbf{NP} \cap \mathbf{co\text{-}NP}$$

Other known facts:

1. TAUT $\in$ **co-NP**. It is not known whether TAUT $\in$ **NP**.
2. Primes $\in$ **co-NP** (trivial), and Primes $\in$ **NP** (Pratt 1975).
3. In fact Primes $\in$ **P** (Agrawal/Kayal/Saxena 2002).

4.4 Oracles for P, NP

- We have seen that diagonalization is a powerful method.
- Maybe one day one can settle the $\mathbf{P}$ =$\mathbf{NP}$ question using diagonalization?
- Unfortunately, this is not the case.
- Any proof of $\mathbf{NP} \nsubseteq \mathbf{P}$ by diagonalization relativizes: for all oracles C: $\mathbf{NP}^C \nsubseteq \mathbf{P}^C$.
- One can define oracles A, B such that both $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^A \neq \mathbf{NP}^A$.

The idea of an *Oracle* TM has already been introduced in Definition 2.84.

Definition 4.27 ($\mathrm{P}^A, \mathrm{NP}^A$)

Let $A \subseteq \Sigma^*$ be any language.

- P^A: P^A is the set of all languages that are accepted by a *DTM with oracle A* in polynomial time.
- NP^A: NP^A is the set of all languages that are accepted by a *NTM with oracle A* in polynomial time.

Observation

If *diagonalization* can be used to show $\mathrm{NP} \nsubseteq \mathrm{P}$, then for each language A: $\mathrm{NP}^A \nsubseteq \mathrm{P}^A$.

$\rightsquigarrow$ **blackboard 4.4**

Is there an oracle A such that $\mathbf{P}^A$ contains **undecidable** languages? $\leadsto$ **exercise**

Theorem 4.28 (Baker/Gill/Solovay)

There exist oracles A, B with:

- $\mathbf{P}^A = \mathbf{NP}^A$,
- $\mathbf{P}^B \neq \mathbf{NP}^B$.

Proof.

$\mathbf{P}^A = \mathbf{NP}^A$: Let A be any $\mathbf{PSPACE}$-complete problem. Then

$$\mathbf{NP}^A \subseteq \mathbf{NSPACE} = \mathbf{PSPACE} \subseteq \mathbf{P}^A$$

which implies equality.

$\mathbf{P}^B \neq \mathbf{NP}^B$: For a language B let

$$L(B) = \{x | \exists w\, (|x| = |w| \ \wedge\ w \in B)\}.$$

Obviously, $L(B) \in \mathbf{NP}^B$. We want to construct a B with $L(B) \notin \mathbf{P}^B$.
Enumerate all Oracle TMs $\mathrm{M}_1, \ldots, \mathrm{M}_k, \ldots$ and construct a language B with $L(B) \neq L(\mathrm{M}_k^B)$ by diagonalization. $\rightsquigarrow$ **exercise**

4.5 **PH**: Polynomial Hierarchy

The SAT problem gives rise to a natural extension. Instead of asking for a valuation of the variables, why not asking questions of the form

> *Is there* **for all** x, y *an assignment for* z *such that* $(x \vee y \vee z) \wedge (\neg x \vee \neg z)$ *is true?*

We can write this in more compact form as

$$\forall x \, \forall y \, \exists z \, ((x \vee y \vee z) \ \wedge \ (\neg x \vee \neg z))$$

We define the language L_{QBF} of quantified boolean formulae as follows:

Definition 4.29 (L_{QBF})

Quantified boolean formulae are defined inductively:

- A variable x is a QBF. The occurrence of x is **free**.
- If ϕ, ψ are QBF, then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$. The occurrence of variables in these expressions is the same as in ϕ, ψ.
- If ϕ is QBF, then $\exists x\phi$ and $\forall x\phi$ are as well. The scope of $\forall, \exists$ are all free occurrences of x. The occurrence of x in $\exists x\phi$ or $\forall x\phi$ are **bound**.

Definition 4.30 (Satisfiability: TQBF)

$$L_{\text{TQBF}} := \{\phi : \phi \in L_{QBF} \text{ is a well-formed formula, } \phi \text{ does not have any free variables, } \phi \text{ is true}\}$$

How can we decide whether a given $\phi \in L_{QBF}$ without free variables is true or not?

We systematically get rid of all quantified variables by instantiating them to (all combninations of) true or false (and keeping in mind whether they are universally or existentially quantified). We end up with a (finite) set of boolean formulae that we can all check.

Theorem 4.31 (L_{TQBF} is PSPACE-complete)

The problem whether a QBF is true is PSPACE*-complete.*

Proof.

$\rightsquigarrow$ **blackboard 4.5**

Remember that the original SAT problem concerned arbitrary (existential) formulae. k-SAT required the formulae to be in a certain form (CNF). Can we define an analogue for TQBF?

Definition 4.32 (Satisfiability: QSAT)

$$L_{\text{Qsat}} := \{\langle Q_1 x_1 Q_2 \ldots Q_n x_n, \phi\rangle : \quad \phi \in \Sigma^*_{sat},$$

Q_i alternating quantifiers of the form $\forall$ bzw. $\exists$,

all variables in ϕ are among $x_1, \ldots, x_n$,

such that the formula $Q_1 x_1 Q_2 \ldots Q_n x_n \phi$ is true $\}$

Why do we require that the quantifiers are alternating? Could we also allow formulae of the form:

$$(\forall x \forall y (x \vee \neg y)) \; \wedge \; (\exists z \forall w (z \wedge w))$$

Can we transform any L_{QBF} formula into one of the form required for L_{Qsat}? I.e. does the following hold

$$L_{\mathrm{TQBF}} \preceq_{\mathrm{pol}} L_{\mathrm{Qsat}}?$$

Yes, here is an algorithm ($\leadsto$ **exercise**).

The last algorithm leads to the notion of an *alternating Turing machine*.

Let us consider the acceptance of an NTM. We can view such an NTM as consisting of **existential** states:

- such a state is *accepting*, if *there is at least one* outgoing path that is *accepting*,
- it is *non-accepting*, if *all outgoing paths* are either *rejecting or infinite*.

We are now ready to define the notion of an

Definition 4.33 (Alternating Turing Machine (ATM))

An *alternating Turing machine*, ATM for short, is a NTM where the following holds:

1. each state is either an $\forall$ state, a $\exists$ state or a normal state,
2. $\forall$ and $\exists$ states have exactly two children (outgoing paths),
3. a normal state has exactly one child.

A state s is *accepting*, **iff**

1. s is an $\forall$ state and all its children are accepting states,
2. s is an $\exists$ state and at least one of its children is an accepting state,
3. s is a normal state and its child is accepting.

An ATM is *accepting*, if its start state is accepting.

Definition 4.34 (Polynomial Hierarchy, ATM version)

We define *APTIME* to be the set of all languages that can be recognized by *alternating Turing machines in polynomial time*. We also define:

- Π_i^P: the set of languages that can be recognized by ATM's in polynomial time, where the first special state is an $\forall$ state, and there are at most i changes between the states
- Σ_i^P: the set of languages that can be recognized by ATM's in polynomial time, where the first special state is an $\exists$ state, and there are at most i changes between the states.

Finally: $\mathbf{PH} := \bigcup_{i\in\mathbb{N}} (\Pi_i^P \cup \Sigma_i^P)$

Lemma 4.35 (Relation between NTM's and ATM's)

$\mathbf{NP} = \Sigma_1^P$ *and* $\mathbf{co\text{-}NP} = \Pi_1^P$.

Proof.

We use Theorem 4.21.

$\mathbf{NP} \subseteq \Sigma_1^P$: The ATM guesses the witness c (from Theorem 4.21) and verifies whether $\langle w, c \rangle$ is in L'.

$\Sigma_1^P \subseteq \mathbf{NP}$: The ATM has only $\exists$-states, so it is a NTM.

$\mathbf{co\text{-}NP} = \Pi_1^P$ is similar. □

Analogously to the sequential classes, $\mathbf{APTIME}(f(n))$ consists of all languages acceptable by an $f(n)$ time bounded ATM.

Theorem 4.36 ($\mathrm{APTIME}(f(n))$ vs. $\mathrm{DSPACE}(f(n))$)

Let $f(n) \geqslant n$.

1. *It holds that* $\mathbf{APTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$
2. *It holds that* $\mathbf{DSPACE}(f(n)) \subseteq \mathbf{APTIME}(f^2(n))$

Proof.

Let $\mathcal{M}$ be a f time-bounded ATM. **We construct a f space-bounded DTM $\mathcal{M}'$ that simulates $\mathcal{M}$ on input w:** it does a depth-first search on the tree of all configurations looking for the accepting states. It only accepts if the root corresponds to an accepting state.

Problem: to store the configuration we need $f(n)$ space, but the depth is also $f(n)$, so this gives $f^2(n)$!!!
To do it in $f(n)$ space, we do not store the configuration itself, but just the indeterministic choice that led to the actual configuration (this is done in constant space).

The second result is proved similar to Savitch's theorem. □

Corollary 4.37 ($\mathbf{PSPACE} = \mathbf{APTIME}$)

It holds that $\mathbf{PSPACE} = \mathbf{APTIME}$*.*

Corollary 4.38 (Relation between ATM's and TQBF)

L_{TQBF} *is* $\mathbf{APTIME}$*-complete.*

Consider our notion of an oracle TM introduced in Definition 2.84 on Slide 421. A TM with a language L as oracle can compute more languages than the TM alone.

Now suppose L is NP-complete. **Then a TM with oracle L can do as much as a TM with any L' oracle, where $L' \in \mathrm{NP}$.**

Consequently we define $\mathrm{NP}^{\mathrm{NP}}$ as the class of languages that can be recognized by a NTM with oracles in NP in polynomial time. Thus we can make a polynomial number of calls to arbitrary NP oracles.

Definition 4.39 (MinDNF)

$$L_{\textsf{MinDNF}} := \{\langle\phi, k\rangle : \ \phi \in \Sigma^*_{sat}, \text{wff}, k \in \mathbb{N}$$
$$\text{there is a } \psi \in \Sigma^*_{sat} \text{ such that}$$
$$|\psi| \leqslant k \text{ and } \psi, \phi \text{ are equivalent } \}$$

MinDNF is certainly in $\mathbf{NP}^{\mathbf{NP}}$ (Oracle TM of Definition 2.84) and in Σ_2^P (Hierarchy of Definition 4.34). In fact, we have the following:

Lemma 4.40

MinDNF is $\mathbf{NP}^{\mathbf{NP}}$*-complete.*

Definition 4.41 (Polynomial Hierarchy, Oracle version)

For a complexity class C we denote by $\mathbf{P}^C$ (resp. $\mathbf{NP}^C$) the class of problems solvable in deterministic polynomial (resp. nondeterministic polynomial) time using C-oracles. Let $\Sigma_0 := \Pi_0 := \mathbf{P}$ and

$$\Sigma_{k+1} := \mathbf{NP}^{\Sigma_k}$$
$$\Pi_{k+1} := \mathbf{co\text{-}NP}^{\Sigma_k}$$
$$\Delta_{k+1} := \mathbf{P}^{\Sigma_k}$$

1. Thus Σ_1 is $\mathbf{NP}$ with queries to a P-oracle, i.e. $\Sigma_1 = \mathbf{NP}$.
2. Similarly we have $\Pi_1 = \mathbf{co\text{-}NP}$ and $\Delta_1 = \mathbf{P}$.
3. A problem is in $\Delta_2 = \mathbf{P}^{\mathbf{NP}}$ if it can be solved in deterministic polynomial time with **subcalls to an $\mathbf{NP}$-oracle**.
4. Although the index is 2, Δ_2 is considered to belong to the first level of the polynomial hierarchy.

1. The second level of this hierarchy consists of Σ_2, Π_2 and Δ_3.
2. Here $\Sigma_2 := \mathbf{NP}^{\mathbf{NP}}$: nondeterministic polynomial time with queries to an $\mathbf{NP}$-oracle.
3. $\Pi_2 := \mathbf{co\text{-}NP}^{\mathbf{NP}}$ and $\Delta_3 := \mathbf{P}^{[\mathbf{NP}^{\mathbf{NP}}]}$.
4. It is immediate that

$$\Sigma_k \cup \Pi_k \subseteq \Delta_{k+1} \subseteq \Sigma_{k+1} \cap \Pi_{k+1}$$

5. It has not yet been proved that the inclusions are proper. That is, it is not known whether the hierarchy collapses at some point or not.

Theorem 4.42 (ATM's and oracle TM's correspond)

The hierarchies introduced in Definition 4.41 on Slide 603 and in Definition 4.34 on Slide 596 are identical.

$\leadsto$ **blackboard 4.6**

$\mathbf{EXP}$ can also be reformulated as the space class $\mathbf{APSPACE}$, the problems that can be solved by an alternating Turing machine in polynomial space. This is one way to see that $\mathbf{PSPACE} \subseteq \mathbf{EXP}$.

Which problems are $\mathbf{PH}$-complete?

Lemma 4.43

1. *If there is a* $\mathbf{PH}$*-complete language, then* $\mathbf{PH}$ *collapses.*
2. *If* $\Sigma_k = \Pi_k$ *for a* $k \in \mathbb{N}$*, then* $\mathbf{PH} = \Sigma_k$ *and the hierarchy collapses.*
3. *If* $\mathbf{P} = \mathbf{NP}$*, then the polynomial hierarchy collapses completely, to the* 0th level*,* $\mathbf{P}$*.*
4. *If* $\mathbf{NP} = \mathbf{co\text{-}NP}$*, then the polynomial hierarchy collapses to the* first level*,* $\mathbf{NP}$*.*
5. *If **Graph Isomorphism** is* $\mathbf{NP}$*-complete, then the polynomial hierarchy collapses to the* second level*,* $\mathbf{NP}^{\mathbf{NP}}$*.*

4.6 Structure of PSPACE

What do we know about $\mathbf{PSPACE}$?

1. $\mathbf{PSPACE}$ is a strict superset of the set of all context-sensitive languages.
2. $\mathbf{PH} \subseteq \mathbf{PSPACE}$, but it is not known whether the containment is proper.
3. If $\mathbf{PSPACE} = \mathbf{PH}$, then the polynomial hierarchy collapses (why?).

We have already considered the **word problem for Typ 1 languages** on Slide 153.

Definition 4.44 (L_{CSG})

$L_{\text{CSG}} := \{\langle G, w\rangle :$ G a contextsensitive grammar
w a word generated by G $\}$

Lemma 4.45 ($\mathbf{CSG} \subsetneq \mathbf{PSPACE}$)

L_{CSG} is $\mathbf{PSPACE}$-complete.

But L_{CSG} is contained in $\mathbf{DSPACE}(n^2)$, the bottom of $\mathbf{PSPACE}$: **a contradiction?**
Compare with Lemma 4.43: here we talk about space, not about time.

Proof.

Let L by any language in **PSPACE**, accepted by a DTM $\mathcal{M}$ in space $p(n)$. We apply padding and define

$$L' = \{y\S^{p(|y|)} : y \in L\}$$

Obviously, L' is **contextsensitive**, so let G be a grammar for it. Then the function

$$f : L \mapsto L_{\mathsf{CSG}}; y \mapsto (G, y\S^{p(|y|)})$$

is a polytime reduction and proves the completeness of L_{CSG} for **PSPACE**. □

We define a generalized version of Tic-Tac-Toe. It is played on an $m \times n$ board. The winner has to produce k crosses (or naughts) in a row.

Definition 4.46 (Generalized $L_{\text{Tic-Tac-Toe}}$)

The problem is to decide whether there is a winning strategy for player 1:

$$L_{\text{TTT}} := \{\langle m, n, k\rangle : m, n, k \in \mathbb{N}, \text{ player 1 has a winning strategy for the Tic-Tac-Toe game played on the } m \times n \text{ board } (k \text{ stones in a row win}).\}$$

Obviously, generalized $L_{\text{Tic-Tac-Toe}}$ is in $\mathbf{DSPACE}(mn)$. In fact, this is no surprise:

Theorem 4.47 (PSPACE completeness)

Generalized $L_{\text{Tic-Tac-Toe}}$ is $\mathbf{PSPACE}$-complete.

$\rightsquigarrow$ **exercise**

We refer to Slide 274 for the Domino problem.

Lemma 4.48 (Quasi-fixed rectangle)

We fix $n \in \mathbb{N}$ and two colors col_1, col_2. We also fix two n-vectors of colors. Is there an $m \in \mathbb{N}$ and a tiling of the $n \times m$ rectangle respecting the two vectors (as upper and lower parts) and the two colors on the two sides? This problem is PSPACE-complete.

4.7 EXPTIME/EXPSPACE

- Obviously, $\mathrm{P} \subsetneq \mathrm{EXP}$ and $\mathrm{P} \subsetneq \mathrm{NEXP}$.
- If $\mathrm{P} = \mathrm{NP}$ then $\mathrm{EXP} = \mathrm{NEXP}$.

Lemma 4.49 (DTM accepting in k steps $L_{\text{DTM accept}}$)

The problem to decide whether a DTM accepts an input in at most k steps:

$$L_{\textit{DTM accept}} := \{\langle \mathrm{M}, w, k\rangle : \ \mathrm{M} \textit{ a DTM }, k \in \mathbb{N}, \ w \textit{ the input for } \mathrm{M}, \ \mathrm{M} \textit{ accepts } w \textit{ in at most } k \textit{ steps}\}$$

is EXP-complete.

Proof.

EXP: It is in $\mathbf{EXP}$ because a trivial simulation requires $O(k)$ time, and the input k is encoded using $O(\log k)$ bits.

Complete: It is $\mathbf{EXP}$-complete because, roughly speaking, we can use it to determine if a machine solving an $\mathbf{EXP}$ problem accepts in an exponential number of steps; it will not use more.

- The game of Checkers (“English Draughts”) is usually played on a 8×8 board.
- It has been recently solved: with perfect play, it ends in a draw.
- However, it can be easily generalized to a $n \times n$ board.

Lemma 4.50 ($n \times n$ Checkers)

Solving the game of checkers on a $n \times n$ board is $\mathbf{EXP}$-complete.

The game of Go is also $\mathbf{EXP}$-complete.

Here are two problems that turn out to be complete for $\mathbf{NEXP}$ and $\mathbf{EXPSPACE}$.

Definition 4.51 (Regular expressions L_{regex})

The problem is to decide whether two regular expressions represent the same languages:

$$L_{\text{regex}} := \{\langle r_1, r_2\rangle : \quad r_1, r_2 \text{ regular expressions },\ \mathfrak{I}(r_1) = \mathfrak{I}(r_2)\}$$

$$L_{\text{regex, no } *} := \{\langle r_1, r_2\rangle : \quad r_1, r_2 \text{ not containing } *,\ \mathfrak{I}(r_1) = \mathfrak{I}(r_2)\}$$

While the first problem is $\mathbf{EXPSPACE}$-complete, the second is $\mathbf{NEXP}$-complete.

We refer again to Slide 274 for the Domino problem.

Lemma 4.52 (Square Domino)

We fix a n_0-vector of colors and a particular color col. Is there $n \in \mathbb{N}$ and a tiling of the $n \times n$ square that respects the n_0-vector (as initial segment of the upper side and color col on all the remaining boundary? This problem is $\mathbf{NEXP}$-complete.

In the following we introduce first-order theories of several interesting structures. **A knowledgeable reader can switch to Slide 632.**

- Given a language $\mathcal{L}$ consisting of constants, function- and predicate symbols, we define the set of $\mathcal{L}$-terms and $\mathcal{L}$-formulae (and, most importantly, of $\mathcal{L}$-sentences).
- We then consider the set of all $\mathcal{L}$-sentences that are true in a certain model: the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$ or the reals $\mathbb{R}$.
- These sets are also called theories, or, more precisely, $\mathcal{L}$-theories.

Language

What is the influence of the language $\mathcal{L}$? Which theories are r.e. or even recursive and what is their complexity? How can one prove (un-)decidability or precise (lower bounds) for decidable theories?

In the following definition, we need a base language, or **signature $\mathcal{L}$**. Such a signature consists in general of a set of

- constants,
- function symbols, and
- predicate symbols.

In particular, we consider $\mathcal{L}$ consisting of

Constants: 0, 1,

Functions: binary function symbol $+$,

Predicates: binary predicates $\doteq$, and $<$.

The following definitions define the set of $\mathcal{L}$-terms, as well as the set of first-order sentences that we can interpret in $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{Q}$ or other interesting structures.

Definition 4.53 (First order logic $\mathcal{L}_{FO}$)

The language of first order logic wrt. a signature $\mathcal{L}$ is defined as follows:

- $x, y, z, x_1, x_2, \ldots, x_n, \ldots$: a countable set Var of variables,
- the predicate symbols of the signature $\mathcal{L}$, and the 0-ary predicates **true** and **false**,
- the constants and the function symbols of the signature $\mathcal{L}$,
- $\neg$, $\wedge$, $\vee$, $\rightarrow$: the sentential connectives,
- (,): the parentheses,
- $\forall$, $\exists$: the quantifiers.

Definition (continued)

The concept of an $\mathcal{L}$-term t and an $\mathcal{L}$-formula φ are defined inductively:

Term: **$\mathcal{L}$-terms** t are defined as follows:

1. each variable is a $\mathcal{L}$-term.
2. if f^k is a k-dimensional function symbol from $\mathcal{L}$ and t_1, $\dots$,t_k are $\mathcal{L}$-terms, then $f^k(t_1, \dots, t_k)$ is a $\mathcal{L}$-Term.

The set of all $\mathcal{L}$-terms that one can create from the set $X \subseteq$ Var is called $Term_{\mathcal{L}}(X)$ or $Term_{\Sigma}(X)$. Using $X = \emptyset$ we get the set of basic terms $Term_{\mathcal{L}}(\emptyset)$, short: $Term_{\mathcal{L}}$.

Definition (continued)

Formula: $\mathcal{L}$-formulae φ are also defined inductively:

1. if P^k is a k-dimensional predicate symbol from $\mathcal{L}$ and t_1, ...,t_k are $\mathcal{L}$-terms then $P^k(t_1, \ldots, t_k)$ is a $\mathcal{L}$-formula
2. for all $\mathcal{L}$-formulae φ is $(\neg\varphi)$ a $\mathcal{L}$-formula
3. for all $\mathcal{L}$-formulae φ and ψ are $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ $\mathcal{L}$-formulae.
4. if x is a variable and φ a $\mathcal{L}$-formula then are $(\exists\, x\, \varphi)$ and $(\forall\, x\, \varphi)$ $\mathcal{L}$-formulae.

Definition (continued)

Atomic $\mathcal{L}$-formulae are those which are composed according to 1., we call them $At_{\mathcal{L}}(X)$ ($X \subseteq Var$). The set of all $\mathcal{L}$-formulae in respect to X is called $Fml_{\mathcal{L}}(X)$.

Positive formulae ($Fml^{+}_{\mathcal{L}}(X)$) are those which are composed using only 1, 3. and 4.

If φ is a $\mathcal{L}$-formula and is part of an other $\mathcal{L}$-formula ψ then φ is called **sub-formula** of ψ.

Here are a few examples of formulae we can build:

1. $\forall x \exists y\, (y \doteq x + 1)$,
2. $\forall x \forall y\, (y \doteq x \vee \exists z(x < z \wedge z < y) \vee \exists z(y < z \wedge z < x))$,
3. $\exists y \forall x\, (x < y \vee x \doteq y)$,
4. $\exists y \exists x\, [\, (x + y \doteq 14) \wedge (x + x + x + y \doteq 15)]$.

If we have multiplication $\cdot$ available, we could also consider the following formulae

- $\exists x \exists y \exists z\, (x \cdot x \cdot x + y \cdot y \cdot y \doteq z \cdot z \cdot z)$,
- $\exists x\; x \cdot x \doteq 2$,
- $\forall x \exists y_1 \exists y_2 \exists y_3 \exists y_4\; x \doteq y_1 \cdot y_1 + y_2 \cdot y_2 + y_3 \cdot y_3 + y_4 \cdot y_4$.

Obviously, the more expressive the language, the more formulae can be expressed.

Definition 4.54 (Presburger/Peano Arithmetic)

Let $\mathcal{L}_+ = \{0, 1, +, \doteq, <\}$ and $\mathcal{L}_{(+,\cdot)} = \{0, 1, +, \cdot, \doteq, <\}$.

$$Th(\langle \mathbb{Z}, 0, 1, +, \doteq, < \rangle) = \text{ all } \mathcal{L}_+\text{-sentences true in } \mathbb{Z}.$$
$$Th(\langle \mathbb{N}, 0, 1, +, \doteq, < \rangle) = \text{ all } \mathcal{L}_+\text{-sentences true in } \mathbb{N}.$$

These two theories are called Presburger arithmetic.

Peano arithmetic is obtained when we add multiplication.

$$Th(\langle \mathbb{Z}, 0, 1, +, \cdot, \doteq, < \rangle) = \text{ all } \mathcal{L}_{(+,\cdot)}\text{-sentences true in } \mathbb{Z}.$$
$$Th(\langle \mathbb{N}, 0, 1, +, \cdot, \doteq, < \rangle) = \text{ all } \mathcal{L}_{(+,\cdot)}\text{-sentences true in } \mathbb{N}.$$

Definition 4.55 ($\mathbb{R}$ with addition/multiplication)

Let $\mathcal{L}_+ = \{0, 1, +, \doteq, <\}$ and $\mathcal{L}_{(+,\cdot)} = \{0, 1, +, \cdot, \doteq, <\}$.

$Th(\langle\mathbb{R}, 0, 1, +, \doteq, <\rangle) =$ all $\mathcal{L}_+$-sentences true in $\mathbb{R}$.
$Th(\langle\mathbb{Q}, 0, 1, +, \doteq, <\rangle) =$ all $\mathcal{L}_+$-sentences true in $\mathbb{Q}$.

These two theories are called **the reals (resp. rationals) with addition only**.
When we have multiplication, i.e. $\mathcal{L}_{(+,\cdot)}$, we can express much more.

$Th(\langle\mathbb{R}, 0, 1, +, \cdot, \doteq, <\rangle) =$ all $\mathcal{L}_{(+,\cdot)}$-sentences true in $\mathbb{R}$.
$Th(\langle\mathbb{Q}, 0, 1, +, \cdot, \doteq, <\rangle) =$ all $\mathcal{L}_{(+,\cdot)}$-sentences true in $\mathbb{Q}$.

The first theory is called theory of real closed fields.

1. Give a $\mathcal{L}_+$-sentence that distinguishes $\mathbb{Z}$ and $\mathbb{N}$.
2. Give a $\mathcal{L}_+$-sentence that distinguishes $\mathbb{Q}$ and $\mathbb{N}$.
3. Give a $\mathcal{L}_{(+,\cdot)}$-sentence that distinguishes $\mathbb{Q}$ and $\mathbb{R}$.
4. Define a transformation from $\mathcal{L}_+$-sentences ϕ to $\mathcal{L}_+$-sentences ϕ', such that the following holds

$$\phi \in Th(\mathbb{N}) \text{ iff } \phi' \in Th(\mathbb{Z})$$

Thus $\mathbb{N}$ and $\mathbb{Z}$ are very closely related and therefore they are both called Presburger Arithmetic (although they are formally different).

- It is well-known that multiplication can be reduced to addition: $x \cdot (n+1) \doteq x \cdot n + x$, and $x \cdot (0+1) \doteq x$.
- Can we use this and define multiplication $\cdot$ in the language $\mathcal{L}_+$?
- No, this is not possible. Try to construct a formula in the language $\mathcal{L}_+$.
- It is not possible to come up with such a formula, because each formula has a fixed length and can not depend on the value of a variable occurring in it.

The theories we have introduced are quite complicated: arbitrary sentences that can be formulated in the underlying language.

In the following we will show:

1. Presburger arithmetic is recursive.
2. Peano arithmetic is undecidable (this leads to the arithmetical hierarchy introduced in the next section).
3. The theory of reals **with addition only** is recursive. Its nondeterministic time complexity is at least exponential.
4. It remains recursive (unlike the transition from Presburger to Peano arithmetic) when multiplication is added.

An important technique to show decidability of a first-order theory is quantifier elimination.

Definition 4.56 (Quantifier Elimination)

A theory T of $\mathcal{L}$-sentences admits quantifier elimination, if each sentence can be transformed into an equivalent one which does not contain any quantifiers (and hence also no variables).

Importance of Quantifier elimination

- Most theories do not allow QE in their base language.
- All theories allow QE in an extended language (see eg. Henkin's proof of completeness of FOL).
- It remains to find a suitable extended language.

Suppose we have shown that the theory of reals with addition only admits quantifier elimination. **What can we conclude from that?**

- The atomic sentences are very simple: just equalities and inequalities of fixed rational numbers. These are easily decided to be true or false. And thus any disjunction, conjunction negation etc. is also easily detectable.
- Therefore the theory of reals **with addition only** is decidable.
- A complexity bound is obtained by inspection of the complexity of the quantifier elimination method.

The same conclusions can be drawn from quantifier elimination of Presburger arithmetic.

Lemma 4.57 (Quantifier Elimination (QE))

To show that a theory admits QE*, it suffices to show that each formula of the form*

$$\exists x \bigwedge_i \phi_i(x, y_1, \dots y_r)$$

where $\phi_i(x, y_1, \dots y_r)$ are atomic formulae with possibly additional variables y_i, is equivalent to *a formula without the variable x and without any quantifiers.*

Proof.

Use the usual rules for modifying formulae. □

Lemma 4.58 (Reals with addition)

The theory of **reals with addition only** *admits quantifier elimination.*

Proof.

$\leadsto$ **blackboard 4.7**

Corollary 4.59 ($\mathbb{Q}$ and $\mathbb{R}$ are indistinguishable in $\mathcal{L}_+$)

$$Th(\langle \mathbb{Q}, 0, 1, +, \doteq, < \rangle) = Th(\langle \mathbb{R}, 0, 1, +, \doteq, < \rangle).$$

Corollary 4.60 (Axioms for $\mathbb{R}, \mathbb{Q}$ in $\mathcal{L}_+$)

The following is a complete axiomatization *of* $Th(\langle \mathbb{R}, 0, 1, +, \doteq, < \rangle)$ *in the language* $\mathcal{L}_+$. *I.e.* **each sentence of $Th(\langle \mathbb{R}, 0, 1, +, \doteq, < \rangle)$ follows from this axiomatization.**

1. *Axioms expressing that* $<$ *is a* dense linear order without endpoints.
2. *Axioms expressing that* $\langle \mathbb{R}, 0, + \rangle$ *is a* group with neutral element 0.
3. $0 < 1$, $\forall x \, (x < y \; \rightarrow \; x + 1 < y + 1)$.

Lemma 4.61 (Reals with addition and multiplication)

The theory of reals with addition and multiplication admits quantifier elimination.

Proof.

$\rightsquigarrow$ **blackboard 4.8**

Corollary 4.62 (Axioms for $\mathbb{R}$ in $\mathcal{L}_{(+,\cdot)}$)

The following is a complete axiomatization of $Th(\langle \mathbb{R}, 0, 1, +, \cdot, \doteq, < \rangle)$ in the language $\mathcal{L}_{(+,\cdot)}$. I.e. each sentence of $Th(\langle \mathbb{R}, 0, 1, +, \cdot, \doteq, < \rangle)$ follows from this axiomatization.

1. *Axioms expressing that $\mathbb{R}$ is an ordered field.*
2. $\forall x\,(x \geqslant 0 \;\rightarrow\; \exists y\, x \doteq y^2)$.
3. *All polynomials of odd degree have a zero (this is an axiom schema: for each fixed odd order n, we can write down an axiom expressing the existence of a zero).*

Often, it is convenient to extend the language a little and show QE in this extended language.

Lemma 4.63 (Presburger arithmetic)

The theory of Presburger arithmetic admits quantifier elimination in the following language: $0, 1, +, \leqslant, -, \mid, \nmid$, *where* $-$ *is a unary function symbol,* $\mid$ *and* $\nmid$ *are binary predicates between a positive integer* $k \doteq 1 + 1 + \ldots + 1$ *(*k *times) and a term* t*. Equality* $t \doteq t'$ *can be defined by* $t \leqslant t' \ \wedge \ t' \leqslant t$*.*
The unary function symbol satisfies $-t + t \doteq 0$*.*
$k \mid t$ *is defined by* $\exists x\,(t \doteq kx)$*.* $k \nmid t$ *is defined by* $\neg(k \mid t)$*.*

Proof.

$\rightsquigarrow$ **blackboard 4.9**

Corollary 4.64 (Axioms for $\mathbb{N}$ in $\mathcal{L}_+$)

The following is a complete axiomatization of $Th(\langle \mathbb{N}, 0, 1, +, \doteq, < \rangle)$ in the language $\mathcal{L}_+$. I.e. **each sentence of $Th(\langle \mathbb{N}, 0, 1, +, \doteq, < \rangle)$ follows from this axiomatization**. *$Th(\langle \mathbb{N}, 0, 1, +, \doteq, < \rangle)$ is therefore decidable.*

1. $\forall x\, x + 0 \doteq x$,
2. $0 + 1 \doteq 1$, $\forall x \neg 0 \doteq x + 1$,
3. $\forall x \forall y\, x + y \doteq y + x$,
4. $x + 1 \doteq y + 1 \;\rightarrow x \doteq y$,
5. $\forall x \forall y\, (x + y) + 1 \doteq x + (y + 1)$,
6. *Let $\phi(x)$ be a formula in $\mathcal{L}_+$ that contains at most a free variable x. Then the following is an axiom*

$$\phi(0) \;\wedge\; \forall x\, (\phi(x) \rightarrow \phi(x+1)) \;\rightarrow\; \forall y \phi(y).$$

A closer inspection of the QE procedure yields the following results.

Theorem 4.65 (Complexity of Arithmetic and $\mathbb{R}$)

1. *Any decision procedure for Presburger arithmetic requires* at least nondeterministic time $2^{2^{cn}}$.
2. *Any decision procedure for the reals* **with addition only** *requires* at least nondeterministic time 2^{cn}.
3. *Presburger arithmetic can be decided in* $2^{2^{cn}}$ *nondeterministic space and in* $2^{2^{2^{dn}}}$ *nondeterministic time.*
4. *The theory of reals* **with addition only** *can be decided in* 2^{cn} *nondeterministic space and in* $2^{2^{dn}}$ *nondeterministic time.*

5. More advanced topics

5 More advanced topics
- Arithmetical Hierarchy
- Analytical Hierarchy
- Descriptive Complexity

Motivation

In this section we consider several advanced topics. They are all closely related to logics variants of first-order logic (FO) and to second-order logic (SO).
We consider:

1. The *arithmetical hierarchy*, based on the first-order (undecidable but r.e.) theory of the natural numbers.
2. The *analytical hierarchy*, based on the second-order (undecidable and not even r.e.) theory of the natural numbers.
3. *Descriptive complexity*: we describe **complexity classes** by certain (fixpoint) *logics*. More precisely, we show that *problems in certain classes* are exactly those that can be *expressed in certain logics*.

5.1 Arithmetical Hierarchy

The arithmetical hierarchy is based on the theory $Th(\mathbb{N})$ of the natural numbers in the language $\mathcal{L}_{(+,\cdot)}$ (see Definition 4.54).

We note that we can

- eliminate 1 (see next two slides), and
- eliminate $<$: $x < y$ iff $\exists z\ (\neg z \doteq 0\ \wedge\ x + z \doteq y)$,
- so that we can assume wlog. that the language of Peano arithmetic consists of just $\{0, +, \cdot, \doteq\}$.

Later we show that we can also get rid of $\doteq$, at least for our purposes.

An illustrating example

We are now distinguishing between formulae in a certain language (these formulae may contain the symbols $0, +, \cdot$) and their interpretation in a certain structure $\mathcal{N}$: $0^{\mathcal{N}}$ is the neutral element in this structure, $+^{\mathcal{N}}$ is a function on this structure while $0, +$ are just symbols.

Example 5.1 (Natural numbers in different languages)

- $\mathcal{N}_{Pr} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$ (*„Presburger Arithmetic"*),
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \doteq^{\mathcal{N}})$ (*„Peano Arithmetic"*),
- $\mathcal{N}_{PA'} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \doteq^{\mathcal{N}})$ (variant of $\mathcal{N}_{PA}$).

These sets each define the natural numbers, but in different languages.

Question:

If the language is bigger then we can express more.
Is $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$ more expressive then $\mathcal{L}_{\{0,+,\cdot,\doteq\}}$?

Answer:

No, because one can replace the $1^{\mathcal{N}}$ by a $\mathcal{L}_{\{0,+,\cdot,\doteq\}}$-formula: there is a $\mathcal{L}_{\{0,+,\cdot,\doteq\}}$-formula $\phi(x)$ so that for each variable assignment ρ the following holds:

$$\mathcal{N}_{\{0,1,+,\cdot,\doteq\}} \models_\rho \phi(x) \text{ \underline{\textbf{iff}} } \rho(x) = 1^{\mathcal{N}}$$

- Thus we can define a macro for 1.
- Each formula of $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$ can be transformed into an equivalent formula of $\mathcal{L}_{\{0,+,\cdot,\doteq\}}$.

Question:

Is $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$ perhaps more expressive than $\mathcal{L}_{\{0,1,+,\doteq\}}$, or can the multiplication be defined somehow?

We will see later that $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$ is indeed more expressive:

- the set of sentences valid in $\mathcal{N}_{\{0,1,+,\doteq\}}$ is decidable (see Corollary 4.64), whereas
- the set of sentences valid in $\mathcal{N}_{\{0,1,+,\cdot,\doteq\}}$ is not even recursively enumerable.

Arithmetical formulae

1. Recall Definition 2.66 on Slide 371: consider a polynomial $p(x_1, \ldots, x_k)$ in the variables $x_1, \ldots, x_k$ with coefficients in $\mathbb{Z}$.
2. The set of all tuples $\langle n_1, \ldots, n_k \rangle$ of natural numbers that are zeroes of $p(x_1, \ldots, x_k)$ is called an arithmetical predicate.
3. We can rewrite the equation $p(x_1, \ldots, x_k) = 0$ and put all **negative terms on the right hand side**. This results in a $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$-formula that can be evaluated in the natural numbers.
4. We call any $\mathcal{L}_{\{0,1,+,\cdot,\doteq\}}$-formula arithmetical.
5. On the next slide we define a finer grained hierarchy of arithmetical formulae.

Definition 5.2 (Arithmetical Hierarchy)

We call an arithmetical formula Σ_k^0 (resp. Π_k^0) if it is of the form $\exists\forall\ldots\phi$ (resp. $\forall\exists\ \ldots\phi$) where ϕ is a quantifier-free $\mathcal{L}_{\{0,1,+,\cdot,\dot{=}\}}$-formula and there are at most $k-1$ alternations of quantifier-blocks.

We call a set $M \subseteq \mathbb{N}$ of natural numbers Σ_k^0-definable (resp. Π_k^0-definable), if M is definable by a Σ_k^0-formula (resp. Π_k^0-formula). I.e. there is a Σ_k^0-formula (resp. Π_k^0-formula) $\phi(x)$ with one free variable x such that

$$\mathcal{N} \models \phi(i) \text{ iff } i \in M.$$

TU Clausthal
Clausthal University of Technology

- Σ^0_0-definable = Π^0_0-definable = recursive sets.
- Σ^0_1-definable = r.e. sets (Hilbert's 10th problem, see Corollary 2.71).
- Π^0_1-definable = complements of r.e. sets
- An example of a Σ^0_1 definable set: the set $\mathcal{H}$ of all Gödel numbers of those Turing-machines that stop on their own Gödel number.

- The higher a problem lies in the hierarchy, the **more undecidable** it is. For example a problem located at the second level, say Σ_2^0, can be thought of as **being recursively enumerable using an oracle which solves Σ_1^0-problems** (like the halting problem).
- Analogously to the polynomial hierarchy we have the notions of Σ_k^0-complete and Π_k^0-complete. As an example, the **halting problem is Σ_1^0-complete**.

In contrast to the polynomial hierarchy, the **arithmetical hierarchy is strict**. From now on we use Σ^0_{k+1} and Π^0_{k+1} not only as a set of syntactically defined formulae, but also as a **set of arithmetical predicates**, that can be defined by respective formulae.

Theorem 5.3 (Arithmetical Hierarchy is Strict)

We denote by Δ^0_k the intersection of Σ^0_{k+1} and Π^0_{k+1}. We have

1. *For each $k \in \mathbb{N}$: $\Sigma^0_k \backslash \Pi^0_k \neq \emptyset$.*
2. *For each $k \in \mathbb{N}$, $\Sigma^0_k \cup \Pi^0_k \subsetneqq \Delta^0_k = \Sigma^0_{k+1} \cap \Pi^0_{k+1}$.*

Proof.

$\leadsto$ **blackboard 5.1**

- In the rest of this section, we change our treatment of Peano arithmetic slightly.
- We show that we do not need the equality symbol.
- We show that we can consider $+, \cdot$ as predicates, not functions.
- Therefore we need an additional unary function symbol: $s()$, the successor function.
- We can then show that our previous results, which involve the infinite theory $Th(\mathbb{N})$, carry over to a finitely axiomatizable theory PA_{fin}.

Question:

Can $\doteq$ be simulated in "FO without $\doteq$"?

Answer:

We can try to axiomatize $\doteq$. We consider FO without $\doteq$ with an additional binary predicate eq and require the following axioms:

- $\forall x \, eq(x, x)$,
- for each function symbol f with adequate arity:

$$\forall x_1 \ldots x_n, y_1 \ldots y_n \, (eq(x_1, y_1) \wedge \; \ldots \; \wedge eq(x_n, y_n)) \\ \rightarrow eq(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)),$$

- for each predicate symbol P with adequate arity:

$$\forall x_1 \ldots x_n y_1 \ldots y_n \, (eq(x_1, y_1) \wedge \; \ldots \; \wedge eq(x_n, y_n)) \\ \rightarrow (P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n)).$$

Important:

This set of axioms depends on the underlying language $\mathcal{L}$, we call it $\mathrm{EQ}_{\mathcal{L}}$.

Question:

Let ϕ be a formula in FO with $\doteq$ and let $\phi[\doteq/eq]$ be obtained from ϕ by replacing each appearing $\doteq$ with eq.
Does the following equivalence hold?

$$T \models_{\doteq} \phi \text{ iff } T[\doteq/eq] \cup \mathrm{EQ}_{\mathcal{L}} \models \phi[\doteq/eq] \text{ ?}$$

Answer:

Unfortunately not, because:

1. The axioms $\mathrm{EQ}_{\mathcal{L}}$ leave open the possibility, that the interpretation of eq in a structure $\mathcal{A}$ consists of $A \times A$ (thus all elements are equivalent).
2. Models may contain elements which **can not be represented by terms** of the considered language – we can make only limited statements about such elements.

It can be shown that we are not able to completely axiomatize the identity *even with additional axioms*.

We dealt with the natural numbers with addition and multiplication before. There were two problems that we can improve:

- the **set of terms is infinite and quite complicated** and
- there is **no finite axiomatization** (not even for Presburger arithmetic, see Corollary 4.64).

We now look at them differently and consider the language $\mathcal{L}_{PA}$ with

- a constant 0 and a unary function symbol s,
- a binary predicate symbol eq and two ternary predicate symbols $\oplus$ and $\otimes$.

Before defining some axioms about $\oplus$ and $\otimes$ we define the following abbreviation: the formula $\exists! z\, \phi(z)$ stands for

$$\exists z\, (\, \phi(z) \, \wedge \, \forall y\, \phi(y) \rightarrow eq(y, z)).$$

Definition 5.4 (Peano arithmetic PA$_{fin}$)

PA$_{fin}$ consists of EQ$_{\mathcal{L}_{PA}}$ and the following axioms:

$$\begin{array}{ll} \forall x \forall y \exists! z & \oplus(x, y, z) \\ \forall x & \oplus(x, 0, x) \\ \forall x \forall y \forall z & \oplus(x, y, z) \rightarrow \oplus(x, s(y), s(z)) \end{array}$$

$$\begin{array}{ll} \forall x \forall y \exists! z & \otimes(x, y, z) \\ \forall x & \otimes(x, 0, 0) \\ \forall x \forall y \forall z \exists z' & \otimes(x, y, z) \rightarrow (\otimes(x, s(y), z') \land \oplus(z, x, z')) \end{array}$$

In contrast to Definition 4.54, we have the following properties:

- There is no equality symbol with a fixed interpretation (it is just any binary relation satisfying certain axioms).
- The set of terms is quite simple: $s(0), s(s(0)), \ldots$.
- We have a finite axiomatization.

Obviously $\mathcal{N} := (\mathbb{N}_0, 0^\mathcal{N}, s^\mathcal{N}, \oplus^\mathcal{N}, \otimes^\mathcal{N}, eq^\mathcal{N})$ is a model of all these axioms:

- $0^\mathcal{N}$ is the “right” 0,
- $s^\mathcal{N}$ is the successor function,
- $\oplus^\mathcal{N}$ is the addition,
- $\otimes^\mathcal{N}$ is the multiplication of natural numbers (each considered as a relation), and
- $eq^\mathcal{N}$ the identity.

The following holds:

- The set $\{\phi : PA_{fin} \models \phi\}$ is recursively enumerable but not recursive (decidable).
- The set $\{\phi : \mathcal{N} \models \phi\}$ is not even recursively enumerable.

The following is the famous result from Kurt Gödel.

Theorem 5.5 (Gödel)

- *Each set of formulae which contains PA_{fin} and has $\mathcal{N}$ as one of its models is* **not recursive**.
- *Each recursively enumerable set of formulae Φ which contains PA_{fin} and has $\mathcal{N}$ as one of its models is* **incomplete**, *i.e. there is a ψ with: $\mathcal{N} \models \psi$ but $\Phi \not\models \psi$.*
- **$\mathcal{N}$ can never be axiomatized completely.**

TU Clausthal
Clausthal University of Technology

5.2 Analytical Hierarchy

What does the superscript 0 mean in Σ_k^0?

It means that we consider first-order formulae (we do not allow our arithmetical formulae to contain second-order quantifiers).

This remark gives rise to the analytical hierarchy, denoted by Σ_k^1, Π_k^1, where we consider second-order arithmetical formulae.

We only count the number of alternations of the **quantifiers over sets**. So any Σ_k^0-formula is in Σ_0^1, for any k.

- For the arithmetical hierarchy the identity $\Sigma^0_0 = \Sigma^0_1 \cap \Pi^0_1$ holds.
- The analogue for the analytical hierarchy does not hold. A counterexample is given by the theory of the natural numbers $\mathcal{N}$: the set of true sentences in arithmetic is in $\Sigma^1_1 \cap \Pi^1_1$ but not in Σ^1_0. This set is also called hyperarithmetical for obvious reasons.

5.3 Descriptive Complexity

Motivation

In this section we take a completely different look at complexity classes. We try to describe them with certain logics, hence the name descriptive complexity.

- In traditional logic, the set of all models of a sentence or a theory matters: Mod(T). We get completeness and compactness results wrt. appropriate calculi.
- For the existence of a calculus, it is important that infinite models exist! Given a first-order theory T that allows finite models the size of which is not bounded, there must exist an infinite model.

Finite models

- In computer science however, only finite models matter. For example our complexity classes consist of languages and each language consists of finite words (or finite structures).

From strings to structures

How can we encode a finite string over an alphabet as a finite structure over an appropriate signature(in the sense of first-order logic)?

From structures to strings

How can we encode a finite structure over a signature (in the sense of first-order logic) into a finite string over an appropriate alphabet?

Definition 5.6 (Language expressible by a logic)

We say that a *language* L consisting of words over an alphabet Σ *is expressible in a logic with signature* τ, if there is a τ-sentence φ such that the following holds

$$L = \{\mathcal{A} : \ \mathcal{A} \text{ is finite and } \mathcal{A} \models \varphi\}$$

Strictly speaking we have to *encode* the τ-structures as words over Σ. However, this can be done in a straightforward way using a linear order in the structure and appropriate normalization.

Finite models

$\mathbf{NP}$ for example contains the class of all 3-colorable undirected graphs (viewed as a language of finite words). Such **graphs** can be seen as first-order structures consisting of (1) a finite universe, the set of nodes, and (2) a binary relation on them (describing the edges between the nodes):

$$\mathsf{G} = \langle \{1, \ldots, n\}, edge^{\mathsf{G}}(\cdot, \cdot) \rangle$$

Question 1: Is there a first-order sentence φ, such that the finite models of φ are exactly the finite 3-colorable graphs?

That would give us a description of $L_{3\text{-col dg}}$ with purely logical means.

Finite models

As another example we consider the language L_{STCON} on Slide 557. The models here are also graphs (directed), but here we have in addition two explicit nodes s, t given:

$$\mathsf{G} = \langle \{1, \ldots, n\}, s^{\mathsf{G}}, t^{\mathsf{G}}, edge^{\mathsf{G}}(\cdot, \cdot) \rangle$$

And we are interested in those graphs where there is a directed path from s to t.

Question 2: Is there a first-order sentence φ, such that the finite models of φ are exactly the finite directed graphs where s^{G} and t^{G} are connected?

That would give us a description of L_{STCON} with purely logical means.

Finite models

- The answer to question 1 is “No”, but if we allow second order logic, then it is “Yes”.
- The answer to question 2 is also “No”, but if we allow an operator that describes transitive closure, then it is “Yes”.
- What if we stick to classical first-order logic? Then we can describe all constant-depth unlimited fan-in circuits.

FO and $L_{3\text{-col dg}}$

- How to define a 3-coloring of a directed graph?
- It can be described with three unary predicates Col_1, Col_2, Col_3 and the conjunction of the following sentences (we denote it by $\varphi_{3\text{-col}}(Col_1, Col_2, Col_3)$):

$$\forall x\,(Col_1(x) \vee Col_2(x) \vee Col_3(x)) \qquad (3)$$

$$\forall x \forall y\,(\,(Col_1(x) \wedge Col_1(y)) \rightarrow \neg edge(x,y)\,) \qquad (4)$$

$$\forall x \forall y\,(\,(Col_2(x) \wedge Col_2(y)) \rightarrow \neg edge(x,y)\,) \qquad (5)$$

$$\forall x \forall y\,(\,(Col_3(x) \wedge Col_3(y)) \rightarrow \neg edge(x,y)\,) \qquad (6)$$

- Each model $\langle \{1,\ldots,n\}, Col_1^{\mathsf{G}}, Col_2^{\mathsf{G}}, Col_3^{\mathsf{G}}, edge^{\mathsf{G}}(\cdot,\cdot)\rangle$ satisfying $\varphi_{3\text{-col}}(Col_1, Col_2, Col_3)$ is a 3-coloring of G.
- **But this can not be expressed as a property in the language just using $edge(\cdot,\cdot)$.**

SOL and $L_{3\text{-col dg}}$

- We consider the **second order sentence** $\varphi_{\exists SO}$: $\exists Col_1 \exists Col_2 \exists Col_3\, \varphi_{3\text{-col}}(Col_1, Col_2, Col_3)$. Finite models satisfying $\varphi_{\exists SO}$ are exactly the finite 3-colorable graphs!
- This works as well for graphs with a hamiltonian cycle, or a vertex cover etc.
- Given any **existential second order sentence** $\varphi_{\exists SO}$ over a signature and an appropriate finite structure.

  How complex is it to test whether the structure satisfies it or not?

Finite models

- We can guess the existential parts (with a NDTM) and then check the rest. By suitably normalizing the encoding of the structure we can make sure that this check can be done in polynomial time (in the size of the structure).
- So the whole test is in $\mathbf{NP}$!!!
- Therefore $\exists SO \subseteq \mathbf{NP}$.

FO and L_{STCON}

- **How to express in FO that s and t are connected?**
- What by defining a binary relation $conn(x, y)$ by

$$\forall x \forall y \, (edge(x, y) \rightarrow conn(x, y)) \qquad (7)$$

$$\forall x \forall y \, (\, (conn(x, y) \leftrightarrow \exists z \, (conn(x, z) \wedge conn(z, y) \,) \qquad (8)$$

- Unfortunately, this does not imply that $conn$ is always the **smallest** relation satisfying this (i.e. the transitive closure of $edge$). Take any finite graph, let $conn$ by the transitive closure of $edge$ and let $conn^*$ be the relation that is true for all pairs of nodes. Clearly, $conn^*$ also satisfies the two axioms. **There is no set of first-order sentences to distinguish between $conn$ and $conn^*$ in the base language.**
- Intuitively, we want to express that $conn$ should be the smallest relation satisfying the two axioms.

FO and $L_{\textsf{STCON}}$

- We extend first-order logic by a transitive closure operator TC. If r is a binary relation, then TC(r) is the transitive closure of r.
- In the language FO(TC) (but not in FO alone) we can easily express the fact that s and t are connected: TC$(edge)(s, t)$.
- Given any **FO formula with TC** over a signature and an appropriate finite structure.

  How complex is it to test whether the structure satisfies the formula or not?

Finite models

- We have to evaluate formulae involving the TC operator. Whether two elements are in the transitive closure can be **guessed** by a finite path connecting them. This guess can be easily verified by a polynomial number of checks: to store the nodes we only need three counters requiring $\log$ space.
- So the whole test is in $\mathbf{NL}$!!!
- Therefore FO(TC) $\subseteq \mathbf{NL}$.

Theorem 5.7

*Let **AC0** be the set of languages recognized by polynomial-size circuits of bounded depth (and unlimited fan-in AND and OR gates). Then:*
***AC0** corresponds exactly to FO.*

Theorem 5.8 (Fagin (1974))

NP *corresponds exactly to existential second order logic* $\exists$***SO***.

Theorem 5.9

NL *corresponds exactly to existential first order logic with transitive closure* ***FO**(TC)*.

There are many more **logics inbetween first-order and second-order logic** than just **FO**(TC) We consider the following:

- FO plus a **commutative** transitive closure operator: **FO**(CTC).
- FO plus a least fixpoint operator: **FO**(LFP).
- SO + transitive closure: **SO**(TC).
- SO + lfp: **SO**(lfp).

Transitive closure is of course a special case for a least fixed point operator.

Theorem 5.10 (Complexity Classes and their logics)

We have the following correspondences

$$
\begin{array}{ccl}
AC0 & \cong & FO \\
L & \cong & FO(CTC) \\
\mathrm{NL} & \cong & FO(TC) \\
\mathrm{P} & \cong & FO(LFP) \\
\mathrm{NP} & \cong & \exists SO \\
\mathrm{co\text{-}NP} & \cong & \forall SO \\
PH & \cong & SO \\
\mathrm{PSPACE} & \cong & SO(TC) \\
\mathrm{EXP} & \cong & SO(LFP)
\end{array}
$$