Complexity and Computability 2017/2018

Gunnar Brinkmann

September 18, 2017

Contents

1 Introduction **2**

2 Recursive functions **9**
- 2.1 Primitive recursive functions 9
- 2.2 The Ackermann function 23
- 2.3 μ-recursive functions 30

3 Turing machines **33**
- 3.1 The Turing machine 33
- 3.2 Combinations and modifications of Turing machines 43
- 3.3 Simulation of Turing machines 47

4 Computability and not computable functions **57**
- 4.1 Turing computability 57
- 4.2 Church's thesis 68
- 4.3 Not computable functions 68
- 4.4 Decidability 71
- 4.5 Gödelisation 72
- 4.6 The halting problem 75
- 4.7 Recursive enumerability 80

5 The incompleteness of elementary number theory **87**
- 5.1 Proof systems 87
- 5.2 An important example: the proof system Z_E 89
  - 5.2.1 Remarks on the chapter *„The incompleteness of elementary number theory“* 108

6 Complexity classes **109**
- 6.1 NP-complete problems 117

1 Introduction

The field of complexity and computability is situated somewhere in between mathematics and computer science. The same is true for books about this topic: books for computer scientists are often a bit less formal (in my eyes in fact not formal enough) and books for mathematicians are often so formal that computer scientists have problems with them – or at least don't like them.

In this text I will try to find a compromise between *too formal* and *not formal enough*. But there are also parts where it is necessary to be **very** informal in order to make things understandable. In those cases I won't use the terms *Theorem* and *proof* but *Observation* and *reasoning* to emphasize that the way the results are presented is not really mathematically rigid...

This text is an English translation of a Dutch translation and extension of German lecture notes based on a book, that (at least in my opinion) presents complexity theory in a very nice way: *W.J. Paul: Komplexitätstheorie*. Unfortunately it appeared only in German and is out of print. The largest problem of this book was a large number of errors – mostly typo's – some of which were quite important. I hope that all those problems are solved in this version of the lecture notes... On the other hand it is not only possible but almost sure that during the translation/extension process one or the other mistake was introduced – so check carefully whether the arguments given in the text are really valid!

A definition *fixes* the meaning of a word. So a formally correct definition can't be wrong. A Theorem can be wrong even if its form is correct. Nevertheless there are essentially two different sorts of definitions: One sort of definition gives a name to a structure that is precisely defined. An artificial example:

1 **Definition** *A graph is called a B-graph if and only if the number of edges is equal to the number of cycles.*

That's an easy definition and there's no problem with it. But there is also no problem with following definition:

2 **Definition** *A graph is called a B-graph if and only if it contains a Hamiltonian path, but not a Hamiltonian cycle.*

Of course the definitions can't be used together – but each of the definitions can be perfectly used – after all the definition explains exactly what the word B-graph means and then one meaning is as good as the other.

But there is still another sort definition – one where we sometimes think that we can say that it is a wrong definition. An example:

3 Definition *Given k computational tasks $t_1, \ldots, t_k$. Each task t_i can be split into n independent parts that can be run in parallel on different computers. If part j of task i needs $s_{i,j}$ seconds, the whole task t_i can be completed in $s_i = \sum_{j=1}^{n} s_{i,j}$ seconds. So s_i is the total time that t_i needs.*

We call t_m the most expensive task *if and only if for all $1 \leq i \leq k$ we have $s_m \leq s_i$.*

Maybe you will say that this is a wrong definition as it should be the *cheapest* task or that there should be a $\geq$ instead of a $\leq$. But what is the difference with the previous definition – why can't we just fix the meaning of the word we defined?
The problem is similar to the two definitions of the same word that you can't use together. We don't have a real definition of *most expensive task* yet – otherwise we don't need a new one – but we have an **intuition** of what it should mean. So something we would accept is:

4 Definition *Given k computational tasks $t_1, \ldots, t_k$. Each task t_i can be split into n independent parts that can be run in parallel on different computers. If part j of task i needs $s_{i,j}$ seconds, the whole task t_i can be completed in $s_i = \sum_{j=1}^{n} s_{i,j}$ seconds. So s_i is the total time that t_i needs.*

We call t_m the most expensive task *if and only if for all $1 \leq i \leq k$ we have $s_m \geq s_i$. (Here $\geq$ is used instead of $\leq$.)*

But that does not mean that we already had some real definition in our head with which we compare the given definition. Most likely we would also accept the following definition:

5 Definition *Given k computational tasks $t_1, \ldots, t_k$. Each task t_i can be split into n independent parts that can be run in parallel on different computers. For $1 \leq j \leq n$ part j of task i needs $s_{i,j}$ seconds. So if we start all jobs in parallel we have to wait $m_i = \max\{s_{i,j}|_{1 \leq j \leq n}\}$ seconds before all parts of task t_i are done.*

We call t_m the most expensive task *if and only if for all $1 \leq i \leq k$ we have $m_m \geq m_i$.*

So the problem is that for some words we already have an intuition and in spite of the fact that the meaning of a word is only fixed by the definition, in these cases we want the definition to *somehow coincide with* our intuition.

In the case of *minimum* or *maximum* it is not really difficult to write up a formal definition that fulfills these requirements. In some cases (e.g. the minimum of a finite set of numbers) our intuition is in fact so precise that we just have to write it up to get a definition.
In other cases we do have some intuition but it is everything else than easy to get a definition out of it. And a large part of this lecture is about such a word: *computability*.

When we talk about the computability of functions, we only talk about functions with values in $\mathbb{N}$ – including the value 0. Note that if you have arbitrary values in $\mathbb{R}$ you can neither write them up, nor store them on a computer, which justifies to not consider them here. In fact you can describe all functions with values that can be represented in a finite way by just focusing on values in $\mathbb{N}$.
Assume for example that M is the set of all finite texts of letters, digits, end-of-line signs and blanks. Furthermore let $f_M : M \to M$ be any function that modifies the text – e.g. formats the text in a way that no line is longer than 80 signs.
We can just code such a text as a number. Given a text m, each symbol is coded as an 8-bit number – e.g. using the ASCII code. The result is a long series of bits which we can interpret as one large number $c(m)$ in binary representation. On the other hand each number n so that the binary representation has a length that is a multiple of 8 can be interpreted as series of ASCII characters – that is: a text which we can write as $c^{-1}(n)$. Now we define the function $f_{\mathbb{N}}$ by

$$f_{\mathbb{N}}(n) = \begin{cases} 0 & \text{if } n \text{ does not code a text in } M \\ c(f_M(c^{-1}(n))) & \text{otherwise} \end{cases}$$

But watch out: this is just some reasoning why it is OK to restrict the attention to $\mathbb{N}$. It is not a proof at all. It is just about interpreting texts as numbers and the other way around. Interpreting texts in other ways you get not only other functions coding the texts, but the same operations on the text are also represented by other functions between natural numbers.
Furthermore we *code* things without having defined what we really mean by *coding*. And in fact we do some operations during this *coding* and it is a question whether these operations are computable and we don't have a definition of computable yet…

1 Exercise *Given an arbitrary function $f_{\mathbb{Q}} : \mathbb{Q} \to \mathbb{Q}$.*
Give a function $f_{\mathbb{N}}$ that shows that you can interpret this function also as a function $\mathbb{N} \to \mathbb{N}$.

If you accept this reasoning, then even the computations in $\mathbb{R}$ can be somehow included in the concept. After all you never dealt with real numbers that you can't represent in a finite way. You write e.g. e or Π, or $sqrt(2)$ in your computations and accepting these notations, they are just strings – or with other words: natural numbers.

When we are talking about *computations*, the term *algorithm* is always an important keyword. Unfortunately this is also a word for which we have an intuition but no clear definition. An algorithm is an instruction on how to do something – e.g. how to manipulate a text or a number (or a piece of metal, or a machine, or...).
If it is an instruction what to do with a natural number x, then one can say that this algorithm computes a function $f()$ with $f(x)$ the number you get after having completed all the manipulations started on the number x. Something that seems reasonable to require (although this is also something one could discuss...) is that it must be possible to describe an algorithm by a finite text with letters from a finite alphabet (like a computer program). But even requiring so few, we can prove something:

6 Observation *If each algorithm can be described by a finite text with letters from a finite alphabet (so different algorithms have different descriptions), then there are functions $f : \mathbb{N} \to \mathbb{N}$ for which no algorithm exists that computes $f()$.*

Reasoning: In fact that follows very easily: there are simply not enough texts for all functions! The number of functions $f : \mathbb{N} \to \mathbb{N}$ is uncountable. It is easy to see that there are at least as many of these functions as real numbers, but you can also prove this directly by using Cantor's diagonal argument.

On the other hand the set of finite texts is clearly countable: a bijection with the natural numbers can e.g. be constructed by sorting texts according to the length and texts with the same length lexicographically.

So there are more functions than texts and therefore also more functions than algorithms. So there must be functions for which no algorithm exists.

■

This means that if we define exactly what an algorithm is and one of the properties of an algorithm is that it can be described by finite texts from a finite alphabet and afterward define a function to be computable if and

only if an algorithm computing it exists, then we have just proven that uncomputable functions exist.
An easy consequence of Observation 6 is

7 Corollary *There are functions $\mathbb{N} \to \mathbb{N}$ and even $\mathbb{N} \to \{0, 1\}$ that can't be computed by any* **C-**, Java-, Pascal-, Lisp-, etc *program – even not if the computer had an infinite memory and could run arbitrarily long.*

2 Exercise *Assume that we don't necessarily want to compute a function $\mathbb{N} \to \mathbb{N}$ exactly, but that we are satisfied with some* approximation *of the value.*
Decide for each of the following definitions of approximation *whether with the requirements for algorithms given above, all functions $\mathbb{N} \to \mathbb{N}$ can be approximated (that is: an approximating function can be computed) or not.*

- *f' approximates f if for all $n \in \mathbb{N}$ we have $|f(n) - f'(n)| \leq 2$.*
- *f' approximates f if for all $n \in \mathbb{N}$ we have $1/2 \leq (f(n)+1)/(f'(n)+1) \leq 2$.*

3 Exercise *In Observation 6 we had a fixed finite alphabet A for all algorithms. What if we relax this condition?*

- *Assume in the following that we have infinitely many possible alphabets $A_1, A_2, \ldots$ with A_i an alphabet with i elements.*

  Assume that we do not require algorithms to use the same fixed alphabet A, but all we require is that an algorithm must be describable by a text from one of these infinitely many alphabets, but that the alphabets may differ for different algorithms.

  Does the argument in the proof still work or are there now enough finite texts to describe – in principle – as much algorithms as there are functions $\mathbb{N} \to \mathbb{N}$? The question is just about the number – not about whether it is really possible to describe these algorithms.

- *Are there enough finite texts for all algorithms if A is fixed for all algorithms, but A is a countably infinite alphabet with letters $t_1, t_2, \ldots$?*

4 Exercise *Assume that our requirement is to work with a fixed finite alphabet A with $|A| \geq 2$, but we do not require the texts describing an algorithm to be finite.*
Is it then possible to give for each function $\mathbb{N} \to \mathbb{N}$ an algorithm computing it?

If we define what we mean by *computable* and *algorithm* then the meaning of these words is fixed by the definitions. But we must take care that these definitions do reflect our intuition. What would we e.g. expect from an algorithm?

- Algorithms should work in steps.
- The next step may depend only from the *input* and the already completed steps.
- Each step must be *elementary*.
- The algorithm must be describable by a finite text from a finite alphabet.
- It must be clear what the result of the algorithm is.

So it would definitely not be an algorithm if there was a step as *"and then a miracle occurs"* or *"and now compute the unbelievably complicated function $f()$ for which so far nobody has found a way to compute it"* or a step as *"now add one or subtract one"*.
In fact you can find "algorithms" in the literature that do not follow these rules and where some steps are not described in enough detail. There are algorithms for which the author claims that they can be implemented in a way that they have a certain running time, but for which so far nobody succeeded to implement them. So this requirement is not as evident as you maybe thought...

From each definition of computability we expect that some elementary functions (such as $f(x) = 5 \quad \forall x \in \mathbb{N}$ or $f(x) = x^2 \quad \forall x \in \mathbb{N}$) fulfil the requirements of the definition – just like all functions for which we can write a computer program.

5 Exercise *Assume a* real *computer to be given – that is: a computer with a finite memory and a CPU that can have just a finite number of different* states, *like content of the cache, etc. But we do assume that the computer can run an infinite time. As the computer is finite, it can not store arbitrarily large numbers, but that does not mean that it can't write numbers much larger than it can store. Even an easy loop such as*

```
for (i=0;i<X;i++) write(``99999999999999999999'');
```

writes a number of $X * 20$ nines – a number that needs much more memory to be stored than the memory necessary to store X and if you choose X large enough also much larger than the memory for X plus the (finite) amount to

store the program containing that line. So you can easily write numbers that are much too large to store in memory.
That you can even output infinite sequences is also clear. In fact most people have already programmed an infinite loop – most of the time without really wanting to do so.

The task is now to output all natural numbers in increasing order separated from each other by blanks.
So the output must begin `0_1_2_3_4_5......11_12_13.....123_124_125.......`
Give a computer program doing that or a proof that it is not possible.

Now we will make a first attempt to define *computability*.

2 Recursive functions

In order to make things a bit easier, we will allow input values in $\mathbb{N}^r$ instead of just $\mathbb{N}$. Later we will see that this can be simulated with values just in $\mathbb{N}$ – so it is really just to make the proofs and arguments easier.
We do not require in all cases that the functions are defined for all $x \in \mathbb{N}^r$, but if this is the case, we call the function total. If the function is only defined for a subset $S \subseteq \mathbb{N}^r$ we call the function partial. In most cases we emphasize the fact that the function is partial only in cases where $S \neq \mathbb{N}^r$.

2.1 Primitive recursive functions

One way to define what a computable function is, is a *constructive way.* We describe some basis of elementary functions that we definitely want to be considered to be computable and some ways to combine functions that we want to preserve computability. Applying these ways to combine functions to the base functions gives us new – computable – functions.
If in the following we write $\mathbb{N}^r$ without explicitly saying what r is, then we always assume $r \in \mathbb{N}, r > 0$. If $f : \mathbb{N}^r \to \mathbb{N}$ is a function and instead of writing $f(y)$ we write $f(n, x)$, then we assume that $n \in \mathbb{N}$ is the first component of $y \in \mathbb{N}^r$ and that $x \in \mathbb{N}^{r-1}$ is the tuple formed by the remaining $r - 1$ components.

8 **Definition** *The class $\mathcal{P}$ of primitive recursive functions is the smallest class that contains the following three base functions and all functions obtained by applying the rules described in a.) and b.) to functions in $\mathcal{P}$.*

1.) *For all $r, s \in \mathbb{N}$ the constant function*

$$c_s^r : \mathbb{N}^r \to \mathbb{N}, \qquad c_s^r(x) = s \quad \forall x \in \mathbb{N}^r$$

2.) *The successor function*

$$o : \mathbb{N} \to \mathbb{N}, \qquad o(x) = x + 1 \quad \forall x \in \mathbb{N}$$

3.) *For all $i, r \in \mathbb{N}, 1 \leq i \leq r$ the projection onto the i-th component – that is the function*

$$p_i^r : \mathbb{N}^r \to \mathbb{N}, \qquad p_i^r(x_1, \ldots, x_r) = x_i \quad \forall (x_1, \ldots, x_r) \in \mathbb{N}^r$$

a.) *Assume $r, m \in \mathbb{N}$. If $f : \mathbb{N}^r \to \mathbb{N}$ and $g_1, \ldots, g_r : \mathbb{N}^m \to \mathbb{N}$ are contained in $\mathcal{P}$, then also $h : \mathbb{N}^m \to \mathbb{N}$ defined as $h(x) := f(g_1(x), \ldots, g_r(x)) \quad \forall x \in \mathbb{N}^m$.*
We call this substitution.

b.) *Assume $r \in \mathbb{N}$. If $f : \mathbb{N}^r \to \mathbb{N}$ and $g : \mathbb{N}^{r+2} \to \mathbb{N}$ are in $\mathcal{P}$, then also the (unique) function $h : \mathbb{N}^{r+1} \to \mathbb{N}$ with the properties*

b1.) $h(0, x) = f(x) \quad \forall x \in \mathbb{N}^r$

b2.) $h(n+1, x) = g(n, h(n, x), x) \quad \forall n \in \mathbb{N}, x \in \mathbb{N}^r$

We call this <u>*primitive recursion*</u>.

6 Exercise *Here we use some word that is intuitively clear:* the smallest. *But which exact definition of* smaller *is used here? If something is smaller, you must have a total or partial order. What is the order used here?*

We also talk about **the** smallest. But is there some unique smallest element in this order? Of course we have to prove that such a unique smallest element $\mathcal{P}$ exists!

9 Remark *If $\bar{M}$ is a set of sets that satisfy 1,2,3,a,b, then so does $\bigcap_{M \in \bar{M}} M$.*

Proof: Of course the base functions are in each $M \in \bar{M}$, so they are also in $\bigcap_{M \in \bar{M}} M$.

But if $f, g, g_1, \ldots, g_r$ are in each $M \in \bar{M}$, then the functions one gets by applying a.) or b.) are also in each $M \in \bar{M}$ and therefore also in $\bigcap_{M \in \bar{M}} M$.

■

Our set $\mathcal{P}$ is therefore the intersection of all sets that satisfy the properties 1,2,3,a and b of Definition 8 and is therefore uniquely determined.

10 Remark *The function $h()$ in part b.) of Definition 8 is uniquely determined.*

Proof: If h_1, h_2 are two functions satisfying b1.) and b2.), then b1.) implies that $h_1(0, x) = f(x) = h_2(0, x)$ for all $x \in \mathbb{N}^r$.

With induction one can now easily show that $h_1(n+1, x) = g(n, h_1(n, x), x) = g(n, h_2(n, x), x) = h_2(n+1, x)$.

■

So that was easy – but nevertheless it was something one had to take care of. It is easy to introduce errors by defining something that doesn't exist or is not unique in spite of the fact that the definition suggests that it is...
The following remark gives an explicit proof that this smallest class $\mathcal{P}$ exists by describing its elements. This description is also more suitable to prove that a given function is in $\mathcal{P}$ than applying Definition 8.

11 Remark *$\mathcal{P}$ is the set of all functions $f()$ for which a finite sequence $f_1, f_2, \ldots, f_s$ of functions exists so that $f = f_s$ and for all $1 \leq i \leq s$ we have that f_i is a base function or there exist $f_{j_1}, \ldots, f_{j_k}$, $1 \leq j_1, \ldots, j_k < i$ so that f_i can be obtained from $f_{j_1}, \ldots, f_{j_k}$ by substitution or primitive recursion.*
This sequence of functions is called derivation of f.

Proof: Let $\mathcal{P}'$ be the set of all functions for which such a derivation exists. By the definition of $\mathcal{P}$ we have immediately $\mathcal{P}' \subseteq \mathcal{P}$.

But as $\mathcal{P}'$ satisfies 1,2,3,a,b, $\mathcal{P}'$ is one of the sets in the big intersection, so $\mathcal{P} \subseteq \mathcal{P}'$ and therefore $\mathcal{P}' = \mathcal{P}$.

■

7 Exercise *We have seen that the size of sets – or precisely: whether they are countable or not – is important when it comes to computability. Is $\mathcal{P}$ countable?*

8 Exercise *Would we get the same class $\mathcal{P}$ if in part 1.) of Definition 8 we would not require that for each $r \in \mathbb{N}$ and each $s \in \mathbb{N}$ the constant function c^r_s is in $\mathcal{P}$ but only for $s = 0$?*

Now we will show that some functions are primitive recursive by explicitly giving a derivation.

1 Example

a.) *The function plus : $\mathbb{N}^2 \to \mathbb{N}$ with plus$(n,m) \to n+m$ is primitive recursive:*

- **1.)** $f_1(n,m,l) := p^3_2(n,m,l)$ *(projection on second component)*
- **2.)** $f_2(n) := o(n)$
- **3.)** *Substitution:*
  $f_3(n,m,l) := f_2(f_1(n,m,l))$ *(so $f_3(n,m,l) = m+1$)*
- **4.)** $f_4(n) := p^1_1(n)$ *(the identity)*
- **5.)** *recursion:*
  $f_5(0,m) := f_4(m)$
  $f_5(n+1,m) := f_3(n, f_5(n,m), m)$

plus$(n,m) = f_5(n,m)$

So we have found a series of functions showing that plus() is primitive recursive.

Instead of plus(n,m) we also write $n+m$.

b.) *The function times* : $\mathbb{N}^2 \to \mathbb{N}$ *with times*$(n, m) \to n \cdot m$ *is primitive recursive:*

From now on we will use functions of which we already showed that they are in $\mathcal{P}$ *in the same way as base functions. Furthermore we will describe the constant base functions just by their value.*

In order to get the real and complete derivation, you just have to take the derivation of the non-base function in the beginning and adapt the indexes of the other functions a bit.

Later on, our proofs that something is in $\mathcal{P}$ *will be even further away from giving an explicit derivation. But you must always be able to give the explicit series* $f_1 = \ldots, \ldots, f_n = \ldots$ *from these proofs in each case – otherwise there is a serious problem...*

1.) $f_1(n, m, l) := p^3_2(n, m, l)$ *(projection on second component)*

2.) $f_2(n, m, l) := p^3_3(n, m, l)$ *(projection on third component)*

3.) $f_3(n, m) :=$ *plus*(n, m)

4.) *Substitution:*
$f_4(n, m, l) := f_3(f_1(n, m, l), f_2(n, m, l))$ *(so* $f_4(n, m, l) = m + l$*)*

5.) $f_5(n) := c^1_0(n)$ *(the constant 0)*

6.) *Recursion:*
$f_6(0, m) := f_5(m)$
$f_6(n + 1, m) := f_4(n, f_6(n, m), m)$

times$(n, m) = f_6(n, m)$

In future we will write that shorter as

times$(0, m) = 0$

times$(n + 1, m) =$ *plus*$($*times*$(n, m), m)$

But you must be aware that the number of parameters as it is used here is not like in the definition. Is that OK? Look at Exercise 16.

Instead of times(n, m) *we also write* $n \cdot m$*.*

c.) *The predecessor function* $v : \mathbb{N} \to \mathbb{N}$ *is defined as*

$$v(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{else} \end{cases}$$

This function is also primitive recursive:

$v(0) = 0$

$v(n+1) = n$

Or one more time in more detail:

$v'(0, m) = c_0^1(m)$

$v'(n+1, m) = p_1^3(n, v'(n, m), m)$

$v(n) = v'(p_1^1(n), c_0^1(n))$

We can use this for

d.) *The* *difference* *or* *minus* *diff* $: \mathbb{N}^2 \to \mathbb{N}$ *is defined as*

$$\mathit{diff}(n, m) = \begin{cases} 0 & \text{if } n \leq m \\ n - m & \text{else} \end{cases}$$

diff() is primitive recursive:

$\mathit{diff}(n, 0) = n$

$\mathit{diff}(n, m+1) = v(\mathit{diff}(n, m))$

Here the recursion is not over the first, but the second parameter. Is that possible for primitive recursive functions? Look at Exercise 16.

We write $n \dot{-} m$ *as a short version of* $\mathit{diff}(n, m)$.

e.) *The* *sign function* *sgn* $: \mathbb{N} \to \mathbb{N}$ *is defined as*

$$\mathit{sgn}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{else} \end{cases}$$

The sign function is primitive recursive:

$\mathit{sgn}(0) = 0$

$\mathit{sgn}(n+1) = 1$

9 Exercise *Prove that the function power* $: \mathbb{N}^2 \to \mathbb{N}$ *defined as* $\mathit{power}(x, y) = x^y$ *is primitive recursive.*
You must give the derivation, but the example functions in Example 1 may be used. The result of $\mathit{power}(0, 0)$ *must be* 1.

12 Definition *A set* $M \subseteq \mathbb{N}^r$ *is called primitive recursive if the characteristic function* $\chi_M : \mathbb{N}^r \to \mathbb{N}$ *defined as*

$$\chi_M(x) = \begin{cases} 1 & \text{if } x \in M \\ 0 & \text{else} \end{cases}$$

is primitive recursive.

13 Lemma *If $M, N \subseteq \mathbb{N}^r$ are primitive recursive sets, then also $M \cap N$, $M \cup N$ and M^c.*

Proof: $\chi_{M\cap N}(x) = \chi_M(x) \cdot \chi_N(x)$

$\chi_{M\cup N}(x) = \operatorname{sgn}(\chi_M(x) + \chi_N(x))$

$\chi_{M^c}(x) = 1 \dot{-} \chi_M(x)$

■

10 Exercise *Prove: If $M, N \subseteq \mathbb{N}^r$ are primitive recursive sets, then also $M \setminus N$ and $M \triangle N$.*

2 Example

- *The set $M_> := \{(n, m) \in \mathbb{N}^2 | n > m\}$ is primitive recursive (analogously for $M_< = \{(n, m) \in \mathbb{N}^2 | n < m\}$):*

  $\chi_{M_>}(n, m) = sgn(n \dot{-} m)$

- *The set $M_= := \{(n, m) \in \mathbb{N}^2 | n = m\}$ is primitive recursive:*

  $M_= = (M_< \cup M_>)^c$

14 Theorem *If $M_1, \ldots, M_k$ are primitive recursive disjoint subsets of $\mathbb{N}^r$ and $f_1, \ldots, f_k$ are primitive recursive functions $\mathbb{N}^r \to \mathbb{N}$ then $g : \mathbb{N}^r \to \mathbb{N}$ defined as*

$$g(x) = \begin{cases} f_1(x) & \text{if } x \in M_1 \\ \vdots & \vdots \\ f_k(x) & \text{if } x \in M_k \\ 0 & \text{else} \end{cases}$$

is also primitive recursive.
So primitive recursive functions allow to make case distinctions.

Proof: $g(x) = \chi_{M_1}(x) \cdot f_1(x) + \cdots + \chi_{M_k}(x) \cdot f_k(x)$

■

The case that the sets are not disjoint and that always the function with the smallest index j is computed for which $x \in M_j$ can be easily simulated by these disjoint sets.

Now we will introduce the $\underline{\text{bounded } \mu\text{-operator}}$. This operator maps a function f to a function $\underline{\mu_b f}$:

15 Definition *For a function $f : \mathbb{N}^{r+1} \to \mathbb{N}$ define the function $\mu_b f : \mathbb{N}^{r+1} \to \mathbb{N}$ as*

$$\mu_b f(n,x) := \begin{cases} min\{m | m \leq n \text{ and } f(m,x) = 0\} & \text{if the set is not empty} \\ 0 & \text{else} \end{cases}$$

Maybe the word *bounded μ-operator* sounds as if this must be something really complicated, but that is not the case: if you apply this operator to a function $f(n,x)$ then the result is (approximately) a function that gives you the first zero (point) of $f(m,x)$ with $m \leq n$. So x is fixed and the first parameter is smallest possible, but in the interval $[0,n]$. It's about searching for zeros – but it is important that this search is just in some region bounded by the first parameter.

16 Theorem *If $f : \mathbb{N}^{r+1} \to \mathbb{N}$ is primitive recursive, then also $\mu_b f$.*

Proof: $\mu_b f(0,x) = 0$

$$\mu_b f(n+1,x) = \mu_b f(n,x) + \\ ((1 \dot{-} \operatorname{sgn}(\mu_b f(n,x))) \cdot \operatorname{sgn}(f(0,x)) \cdot (1 \dot{-} \operatorname{sgn}(f(n+1,x)))) \cdot (n+1)$$

■

11 Exercise *Are all the sgn in the previous proof really necessary?*

17 Corollary *If $M \subseteq \mathbb{N}^{r+1}$ is primitive recursive then also the functions*

- $$\min_M(n,x) := \begin{cases} \min\{m | (m,x) \in M \text{ and } m \leq n\} & \text{if the set is not empty} \\ 0 & \text{else} \end{cases}$$

- $$\max_M(n,x) := \begin{cases} \max\{m | (m,x) \in M \text{ and } m \leq n\} & \text{if the set is not empty} \\ 0 & \text{else} \end{cases}$$

Proof: $\min_M(n,x) = \mu_b \chi_{M^c}(n,x)$

$$\max_M(0,x) = 0$$

$$\max_M(n+1,x) = ((n+1) \cdot \chi_M(n+1,x)) + (1 \dot{-} \chi_M(n+1,x)) \cdot \max_M(n,x)$$

■

With these tools we can now easily show that the whole part and the rest when dividing numbers from $\mathbb{N}$ are primitive recursive too:

18 Theorem *The functions $div : \mathbb{N}^2 \to \mathbb{N}$ and $rest : \mathbb{N}^2 \to \mathbb{N}$ defined as*

$$div(n,m) := \begin{cases} \lfloor n/m \rfloor & \text{if } m \neq 0 \\ 0 & \text{else} \end{cases}$$

and

$$rest(n,m) := \begin{cases} n - (m * \lfloor n/m \rfloor) & \text{if } m \neq 0 \\ n & \text{else} \end{cases}$$

are primitive recursive.
For $div(x,y)$ we also write x/y.

Proof:

$$\mathrm{div}(n,m) := \begin{cases} \max\{l | l \leq n \text{ and } l \cdot m \leq n\} & \text{if } m \neq 0 \\ 0 & \text{else} \end{cases}$$

and

$\mathrm{rest}(n,m) = n \dot{-} (m \cdot \mathrm{div}(n,m))$

■

12 Exercise *The "*max*" in the previous proof does not have exactly the same form as the "*$\max_M$*" in Corollary 17. Define M and write the proof in a way so that it uses exactly the form given in Corollary 17.*

13 Exercise

$$\log^*(n,m) := \begin{cases} \lceil \log_n(m) \rceil & \text{if } n \geq 2, m \geq 1 \\ 1 & \text{if } n = m = 1 \\ 0 & \text{else} \end{cases}$$

Show that $\log^*$ *is primitive recursive without using the* min *and* max *functions but by explicitly using the bounded μ-operator.*

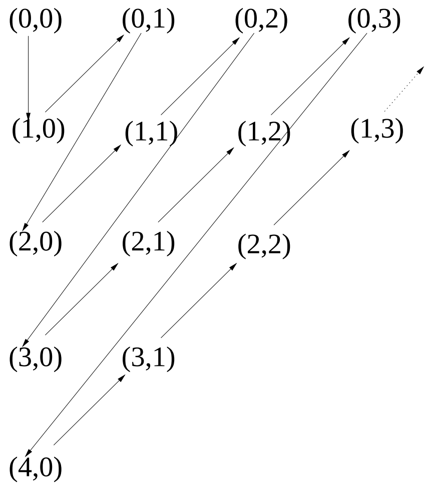
In the beginning we talked about coding strings in natural numbers and we also had an exercise where two natural numbers had to be coded in one. But as we had no concept of computability yet, we could not prove that these coding functions were computable.
Now we will see that you can indeed code vectors of numbers from $\mathbb{N}$ as one single number and can also decode that number to regain the vector – and all that in a primitive recursive way.

19 Theorem *For all $k \in \mathbb{N}$ there are primitive recursive functions with the following properties:*

- *bijective $b^{(k)} : \mathbb{N}^k \to \mathbb{N}$ and*
- *$b_i^{(k)} : \mathbb{N} \to \mathbb{N}$ for $1 \leq i \leq k$ with the property that $b_i^{(k)}(b^{(k)}(n_1, \ldots, n_k)) = n_i$*

One could use this theorem as an argument that in the future it is sufficient to talk about functions $\mathbb{N} \to \mathbb{N}$ as a function $\mathbb{N}^r \to \mathbb{N}$ is primitive recursive if and only if a function $\mathbb{N} \to \mathbb{N}$ obtained by one of these bijections is primitive recursive. But in fact this would make things more difficult instead of easier as already in the derivations functions with more than one parameter occur in a very natural way.

Proof: For $k = 2$ we will use the same technique that is also used to show that $\mathbb{N}^2$ or $\mathbb{Q}$ are countable:



If you write this as a formula, you get

$b^{(2)}(n, m) := (\sum_{i=1}^{n+m} i) + m = \frac{1}{2}(n + m)(n + m + 1) + m$

and that is obviously primitive recursive.

Now we define

$s(l) := \max\{i | i \leq l \text{ and } \frac{1}{2}i(i + 1) \leq l\}$. Then we have $s(b^{(2)}(n, m)) = n + m$ and $s()$ is also primitive recursive.

Let now

$b_2^{(2)}(l) := l \dot{-} \frac{1}{2}s(l)(s(l) + 1)$

and
$b_1^{(2)}(l) := s(l) - b_2^{(2)}(l)$
Substitution gives then
$b_2^{(2)}(b^{(2)}(n,m)) = m$ and therefore $b_1^{(2)}(b^{(2)}(n,m)) = n$.

For $k > 2$ we define the function inductively
$b^{(k)}(n,x) := b^{(2)}(n, b^{(k-1)}(x))$
and
$b_1^{(k)}(l) := b_1^{(2)}(l)$
$b_i^{(k)}(l) := b_{i-1}^{(k-1)}(b_2^{(2)}(l))$ for $2 \leq i \leq k$.

■

Using this you can also prove that functions obtained by simultaneous recursion are primitive recursive. Or to be precise:

20 **Theorem** *If $f_0, f_1 : \mathbb{N}^r \to \mathbb{N}$ and $g_0, g_1 : \mathbb{N}^{r+3} \to \mathbb{N}$ are primitive recursive, then also*
$h_0, h_1 : \mathbb{N}^{r+1} \to \mathbb{N}$ that are for $i = 0, 1$ defined as
$h_i(0,x) = f_i(x)$
$h_i(n+1,x) = g_i(n, h_0(n,x), h_1(n,x), x)$
We call this simultaneous recursion.

14 **Exercise** *Prove Theorem 20.*
Tip: use Theorem 19.

If one would have defined primitive recursive functions allowing functions that go to $\mathbb{N}^r$ instead of just $\mathbb{N}$, the simultaneous recursion in Theorem 20 could be written as a normal recursion. So in a way this theorem says that the seemingly more general case of allowing $\mathbb{N}^r$ (in the case of the theorem $r = 2$) as image space would – up to coding – give the same primitive recursive functions as just allowing $\mathbb{N}$.
Of course one can invent other forms of recursion that – at least at first sight – don't look like primitive recursion and where it is hard to see whether they would extend the class of primitive recursive functions or – like simultaneous recursion – not.

3 **Example** *The following function $\mathbb{N}^2 \to \mathbb{N}$ is another example of another sort of recursion:*

- $f(0, m) := m + 1$
- $f(n + 1, 0) := f(n, 1)$
- $f(n + 1, m + 1) := f(n, f(n + 1, m))$

We have seen that the approach with primitive recursive functions is already pretty strong and one could assume that it is already a good approach to capture our intuitive notion of *computability*. But the following observation says otherwise...

When we use the term normal computer we assume a computer that is – as usual – operated by a finite program but has – different from a usual computer – no limitations on memory or time. So it can run arbitrarily long and the numbers can become arbitrarily large. But it does in fact describe our *idea* of a normal computer. The limitation on memory and time are no limits given by the *principle of a computer* but rather limitations of a given machine.
Of course all this is very informal as we haven't described what a processor can do and which operations are possible. But nevertheless the following observation will shed some light on our approach:

21 Observation • *Primitive recursive functions can be computed with a normal computer.*

- *A normal computer can compute some functions that are not primitive recursive.*

This implies that the class $\mathcal{P}$ is a strict subset of the class of functions that can be computed by a normal computer.

Reasoning: The first part is for people who can program surely obvious as all base functions and all ways to combine functions can be implemented on a normal computer.

The second part is a bit more complicated to prove:

Our arguments need that a normal computer can read numbers and can also fulfil the following tasks:

- If a derivation of a primitive recursive function is given, then the computer can use this derivation to compute the function value for a given number x. This is not exactly the same as our first point – this is a bit stronger and goes into the direction of a *universal* *computer* that gets the derivations as a program, so that one **fixed** program for a computer can – on input an arbitrary derivation – compute the function given by that derivation.

- A computer can test whether a given text (e.g. a sequence of ASCII signs) represents the derivation of a primitive recursive function (that is: parse the text). Of course there are several possible ways to fix how a derivation must be represented as a text. All we need is that there is at least one way to fix that in a way that the computer can check whether it is a valid derivation.
- A computer can generate all finite texts with increasing length and texts with the same length in lexicographic order.

Now we will describe a function of which we can prove that it is not primitive recursive:

Input is a natural number n. If $n = 0$, output 0.

Otherwise the computer starts to generate all finite texts and each time a new text is ready, it is tested whether it represents a derivation of a primitive recursive function. If the computer finds the i-th valid derivation it gets number i. The function that is represented by this derivation is called $f_i()$. Of course different derivations – in fact even infinitely many – can represent the same function.

If the input was n, the computer produces derivations until derivation number n is found. Then it applies the derivation and computes $f_n(n)$. If $f_n(n) = 1$ it outputs 0, otherwise 1.

This defines a function $c : \mathbb{N} \to \mathbb{N}$.

Assume now that $c()$ is primitive recursive. Then there is at least one derivation for $c()$. Assume that the number of that derivation is k, so $c() = f_k()$, which means that $c(i) = f_k(i)$ for all $i \in \mathbb{N}$ and especially $c(k) = f_k(k)$. But if $c(k) = 1$ then $f_k(k) = 0 \neq 1$. So we must have $c(k) \neq 1$, but then $f_k(k) = 1$, so again $c(k) \neq f_k(k)$. This is a contradiction, so our assumption that $c()$ is primitive recursive must be wrong.

■

We will see such an argumentation more often in this lecture. This kind of argument is called a diagonal argument.

What we really needed was not the precise definition of $\mathcal{P}$ but only:

(1) that we can use derivations to determine the result of the derived function

(2) that we can recognise a derivation

(3) that we can sum up derivations

That means that if we added new rules – e.g. new ways of recursion like that in Example 3 – or new base functions that a computer can compute, we can still use the same argumentation as long as the new rules and functions allow (1),(2), and (3).

15 Exercise *Of course everybody knows that C is a better programming language than Java (or not?), but we can even prove this. Discuss this proof and the next theorem. If you think that the proof is wrong, say exactly where the problem is!*

Theorem(?): *Each function that can be computed by a Java-program can also be computed by a C-program, but there are functions that a C-program can compute that can not be computed by any Java-program.*

Proof: *There are Java-to-C compilers (and they can of course be written in C), so it is obvious that for each function for which a Java program exists also a C-program exists: just take the Java program and translate it to C.*

For the second part we work as follows: Write a C-program that sums up all finite texts. Each time a new text is ready, the text is parsed (by a C-program) to judge whether it is a valid Java-program. The function computed by the i-th valid Java-program is called $f_i()$. If the input of the C-program is $n > 0$, we generate texts until we have found the n-th Java program, translate it to C (or interpret it by means of a C program) and apply it to the number n. If the result is 0 we output 1, otherwise (also if the program throws an exception or whatever) we output 0. Call the function computed this way $c()$.

Just like in the proof of Observation 21 we can now conclude that $c(n) \neq f_n(n)$ for all $n \in \mathbb{N}$ and therefore that $c() \neq f_k()$ for each $k \in \mathbb{N}$ – which means that there is no Java program computing $c()$.

■

16 Exercise *As a repetition now some relatively easy exercises that are nevertheless very useful and helped already to represent things in an easier way. Hopefully these exercises also help to get a better intuition for the concept of primitive recursion:*
Assume $r \in \mathbb{N}, r > 0$

(i) *Prove: If* $f : \mathbb{N}^r \to \mathbb{N}$ *and* $g : \mathbb{N}^{r+1} \to \mathbb{N}$ *are in* $\mathcal{P}$, *then also* $h : \mathbb{N}^{r+1} \to \mathbb{N}$ *defined as*

c1.) $h(0, x) = f(x) \quad \forall x \in \mathbb{N}^r$

c2.) $h(n + 1, x) = g(h(n, x), x) \quad \forall n \in \mathbb{N}, x \in \mathbb{N}^r$

and if the function $g' : \mathbb{N} \to \mathbb{N}$ *is in* $\mathcal{P}$, *then also* $j : \mathbb{N}^{r+1} \to \mathbb{N}$ *defined as*

c1.) $j(0, x) = f(x) \quad \forall x \in \mathbb{N}^r$

c2.) $j(n + 1, x) = g'(j(n, x)) \quad \forall n \in \mathbb{N}, x \in \mathbb{N}^r$

(ii) *Prove: if* $\pi : \{1, \ldots, r\} \to \{1, \ldots, r\}$ *is bijective (that is: a permutation), then* $f : \mathbb{N}^r \to \mathbb{N}$ *is primitive recursive if and only if* $g : \mathbb{N}^r \to \mathbb{N}$ *defined as* $g(x_1, \ldots, x_r) := f(x_{\pi(1)}, \ldots, x_{\pi(r)})$ *is primitive recursive.*

This means that recursion must not always be over the second parameter and that also when working with substitution you may interchange positions.

17 Exercise *Assume that* $r \in \mathbb{N}, r > 0$ *and* $f, g : \mathbb{N}^r \to \mathbb{N}$ *are functions with* $f()$ *primitive recursive and* $f(x) \neq g(x)$ *for only finitely many* $x \in \mathbb{N}^r$. *Prove: Then* $g()$ *is also primitive recursive.*
Can you formulate and prove a more general version of Theorem 14 so that you can apply that?

18 Exercise *Prove:*
If $f_0, f_1 : \mathbb{N}^r \to \mathbb{N}$ *en* $g : \mathbb{N}^{r+3} \to \mathbb{N}$ *are primitive recursive, then also* $h : \mathbb{N}^{r+1} \to \mathbb{N}$ *with*
$h(0, x) = f_0(x)$
$h(1, x) = f_1(x)$
$h(n + 1, x) = g(n, h(n, x), h(n - 1, x), x)$ $n \geq 1$

2.2 The Ackermann function

Our Example 3 for another kind of recursion is known as the Ackermann function.

22 Definition *The definition of the Ackermann function $a : \mathbb{N}^2 \to \mathbb{N}$ is*

- $a(0, m) := m + 1$
- $a(n + 1, 0) := a(n, 1)$
- $a(n + 1, m + 1) := a(n, a(n + 1, m))$

This function looks much less artificial than the function $c()$ constructed in the proof of Observation 21. Nevertheless we will see that this function is also **not** primitive recursive.

19 Exercise *In order to get used to this nested recursion – and a corresponding nested induction in some proofs we will first check whether the function is in fact a well defined function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. In the recursion, the result of $a()$ is already used and we must make sure that it is always in $\mathbb{N}$ and can't be negative.*
Prove: for all $n, m \in \mathbb{N}$ we have $a(n, m) \geq 1$.

20 Exercise *At first sight it isn't even obvious that the recursion ever stops. It is easy to* define *recursions that are just not well defined. E.g.:*

- $a(0, m) := m + 1$
- $a(n + 1, m) := a(n, a(n + 1, m + 1))$

Prove that the lexicographic order of $\mathbb{N}^2$ has the property that there are no infinite strictly decreasing sequences. Prove this directly or by proving that it is a well quasi ordering.
Are the parameter pairs in the definition of the Ackermann function strictly decreasing?

If we simply added this kind of recursion to the definition of primitive recursion, then the argumentation in Observation 21 would still work, so we still wouldn't have a satisfactory definition of computability.
But the Ackermann function can obviously *in principle* be computed by a modern computer – if it just has enough time and memory.
If you replace the data types *long int* by types allowing arbitrarily large numbers, the following C-program would already do that:

```
#include<stdlib.h>
#include<stdio.h>

long int a(long int n, long int m)
{
  if (n==0) return m+1;
  if (m==0) return a(n-1,1);
  return a(n-1,a(n,m-1));
}

main(int argc, char*argv[])
{
  long int n,m;

  n=atol(argv[1]); m=atol(argv[2]);

  printf("a(%ld,%ld)=%ld \n",n,m,a(n,m));
}
```

But if you implement and run this program, you will see that already for small values of x, y the computer will become **very** slow and reach its limit. If you run the program on an Intel Core2 Quad CPU Q8200 with 2.33GHz, you need less than 0.001 seconds to compute $a(1,1) = 3$, $a(2,2) = 7$ and $a(3,3) = 61$. Things get bad as soon as the first parameter is 4: for $a(4,1) = 65533$ you need 3.2 seconds, for $a(4,2) = 2^{65536} - 3$ (so written out, this number would fill about 7 pages of these lecture notes) you simply don't want to wait. We have $a(5,0) = a(4,1)$, but that's all that can be done for 5 – let alone for any larger values!
The Ackermann function grows **extremely** fast – and that is also valid for the time necessary to compute it or even just write the digits of the result.
First we will try to find out something about the function $a(n,m)$ for small n.

- $a(0,m) = m + 1$ – that's just the definition.
- $a(1,m) = m + 2$:
  $a(1,0) = a(0,1) = 2$, so it holds for $m = 0$
  if it is true for $0 \ldots m - 1$ then
  $a(1,m) = a(0, a(1,m-1)) = a(1,m-1) + 1 = (m+1) + 1 = m + 2$

- $a(2,m) = 2 \cdot m + 3$:
  $a(2,0) = a(1,1) = 3$, so it holds for $m = 0$
  if it is true for $0 \dots m-1$ then
  $a(2,m) = a(1, a(2, m-1)) = (2 \cdot (m-1) + 3) + 2 = 2 \cdot m + 3$.

- $a(3,m) = 2^{m+3} - 3$:
  $a(3,0) = a(2,1) = 5$, so it holds for $m = 0$
  if it is true for $0 \dots m-1$ then
  $a(3,m) = a(2, a(3, m-1)) = 2 \cdot (2^{m+2} - 3) + 3 = 2^{m+3} - 3$

So for fixed small n you can also find faster algorithms to compute $a(n,m)$.

21 Exercise *Prove:* $a(4,m) = 2^{(2^{(\cdot^{\cdot(2^2)})})} - 3$ *with* $m+3$ *twos in the tower of twos.*

23 Theorem *The Ackermann function is not primitive recursive.*

Proof: First we will give a rough outlook on how the proof works:

We will show that the Ackermann function grows faster than **any** primitive recursive function. Or to be a bit more precise: we will show that any primitive recursive function $f(m)$ is (in a certain way) bounded above by $a_n(m) := a(n,m)$ for some fixed n. This value n does depend on $f()$ though. We will also show that the function $a(m,m)$ is not bounded by any $a_n(m)$, so it can't be primitive recursive.

First some easy results that we will need in what follows:

(i) $a(n,m) > m$

For $n = 0,1,2,3$ and arbitrary m we have already proven that or it follows directly from the definition. For $m = 0$ it follows directly from Exercise 19.

So (i) is valid for all pairs $(n,0)$ and all pairs $(0,m)$ (and also for $(1,m), (2,m), (3,m)$, but we won't need that).

The induction is now about all pairs (n,m) in lexicographic order. Assume that (i) has been proven for all (n',m') that are lexicographically smaller than (n,m) or with one of the parameters 0 (a case for which we have already proven it).

Something special for this induction is surely that already in the first step you prove it for infinitely many cases and not – like in most induction proofs – just for a single case. Furthermore there are also infinitely many smaller pairs for each pair (n, m) with $n > 0$: all pairs (n, m'), with $m' < m$ and all pairs (n', m'') with $n' < n$ and arbitrary m''. So think carefully about whether this induction is really OK. This will also be discussed in an exercise.

So we have $a(n, m) = a(n-1, a(n, m-1)) > a(n, m-1) > m-1$ and as we have two times "'>"' for integer values, this implies $a(n, m) > m$.

Note that the induction step only works if both parameters are larger than 0! For $(n, 0)$ and $(0, m)$ you can't prove (i) by induction over smaller pairs. This was the reason why the cases $(n, 0)$ and $(0, m)$ were handled separately in the beginning.

(ii) $a(n, m+1) > a(n, m)$

$a(0, m+1) = m+2 > m+1 = a(0, m)$

$a(n+1, m+1) = a(n, a(n+1, m)) \overset{(i)}{>} a(n+1, m)$

And this implies immediately

(iii) If $m > m'$ then $a(n, m) > a(n, m')$

(iv) $a(n+1, m) \geq a(n, m+1)$

This can be proven by means of a *normal* induction:

For $m = 0$ we have $a(n+1, 0) = a(n, 0+1)$ – which is just the definition.

So assume that (iv) is valid for $0 \dots m$. Then

$a(n+1, m+1) = a(n, a(n+1, m)) \overset{Ind.,(iii)}{\geq} a(n, a(n, m+1)) \overset{(i),(iii)}{\geq} a(n, m+2)$

and therefore

(v) $a(n+l, m) \geq a(n, m+l)$

(vi) $a(n, m) > n$

The special case $n = 0$ of (v) gives $a(l, m) \geq a(0, m+l) = m+l+1 > l$.

Now we can also prove that $a(n, m)$ is strictly increasing in the first component.

(vii) if $n > n'$ then $a(n, m) > a(n', m)$

If $n = n' + l$ then we have by (v) that $a(n,m) = a(n'+l,m) \geq a(n', m+l) \overset{(iii)}{>} a(n',m)$.

Furthermore

(viii) $a(n+2,m) > a(n, 2\cdot m)$

We prove this by induction over m:

$a(n+2,0) \overset{(vii)}{>} a(n,0)$

Assume that (viii) holds for $0 \dots m$.

$a(n+2,m+1) = a(n+1, a(n+2,m)) \overset{Ind.,(iii)}{>} a(n+1, a(n, 2\cdot m)) \overset{(iv)}{\geq} a(n, a(n,2\cdot m)+1)$

According to (i) we have $a(n,2\cdot m) > 2\cdot m$, so $a(n,2\cdot m)+1 \geq 2\cdot m+2 = 2\cdot(m+1)$.

This result together with (iii) gives $a(n+2,m+1) > a(n, 2\cdot(m+1))$.

Now we need two definitions:

- for $x = (x_1, \dots, x_r) \in \mathbb{N}^r$ let $x_{\max} := \max_{i=1,\dots,r} x_i$
- let B_k be the set of all functions so that there is some $r \in \mathbb{N}, r > 0$ so that $f : \mathbb{N}^r \to \mathbb{N}$ and $f(x) \leq a(k, x_{\max})$ for all $x \in \mathbb{N}^r$.

The intention of what we did so far was to show that for primitive recursive functions one can choose some fixed k so that the function is contained in some B_k:

First we look at the base functions. The base functions $p_i^r(x)$ and $o(x)$ are all bounded by $x_{\max}+1$, so they are in B_0.

According to (vi) we have $a(n,m) > n$, so the constant functions c_s^r are contained in B_s.

Assume now that $h = f(g_1(x), \dots, g_r(x))$ is a function that is formed by substituting $g_1, \dots, g_r$ in f and that $f \in B_{b_f}$ and $g_i \in B_{b_i}$ for $1 \leq i \leq r$. We have to show that $h \in B_{b_h}$ for a certain $b_h \in \mathbb{N}$.

We choose $b_h := \max\{b_f, b_1, \dots, b_r\} + 3$.

For a fixed x and $i \in \{1 \dots r\}$ we define $y_i := g_i(x)$ and $y_{\max} := \max_{i=1,\dots,r} y_i$.

Then $a(b_h, x_{\max}) \overset{(iv)}{\geq} a(b_h - 1, x_{\max} + 1) = a(b_h - 2, a(b_h - 1, x_{\max}))$.

As $b_h - 1 > b_i$, $1 \leq i \leq r$ we have $a(b_h - 1, x_{\max}) \overset{(vii)}{>} a(b_i, x_{\max}) \overset{g_i \in B_{b_i}}{\geq} y_i$

With (iii) we also have $a(b_h - 2, a(b_h - 1, x_{\max})) \geq a(b_h - 2, y_{\max})$ and therefore $a(b_h, x_{\max}) > a(b_h - 2, y_{\max})$

As $b_h - 2 > b_f$, together with (vii) this implies:

$a(b_h - 2, y_{\max}) > a(b_f, y_{\max}) \overset{f \in B_{b_f}}{\geq} f(y_1, \ldots, y_r) = f(g_1(x), \ldots, g_r(x)) = h(x)$ and therefore $h \in B_{b_h}$.

Assume now that $h : \mathbb{N}^{r+1} \to \mathbb{N}$ is defined as $h(0, x) := f(x)$, $h(n + 1, x) = g(n, h(n, x), x)$ with $f \in B_{b_f}$, $g \in B_{b_g}$. Choose $b_h := \max\{b_f, b_g\} + 3$.

A problem is now that we don't know in advance whether the maximum is in the first parameter n or one of the others.

To solve this problem, we will first show that for all $n \in \mathbb{N}$, $x \in \mathbb{N}^r$ we have $a(b_h - 2, x_{\max} + n) > h(n, x)$. We do this by induction in n.

$n = 0$:

For $h(0, x) = f(x)$ we have $a(b_h - 2, x_{\max} + 0) > a(b_f, x_{\max}) \geq f(x) = h(0, x)$.

Assume now that our claim holds for $0, \ldots, n$. Define $m := h(n, x)$, which gives $h(n + 1, x) = g(n, m, x)$.

With induction we have $a(b_h - 2, x_{\max} + n) > m$. Furthermore $a(b_h - 2, x_{\max} + n) > x_{\max} + n \geq \max\{x_{\max}, n\}$ and therefore $a(b_h - 2, x_{\max} + n + 1) = a(b_h - 3, a(b_h - 2, x_{\max} + n)) > a(b_h - 3, \max\{x_{\max}, n, m\}) \geq a(b_g, \max\{x_{\max}, n, m\}) \geq g(n, m, x) = h(n + 1, x)$.

We use that to show that

$a(b_h, \max\{n, x_1, \ldots, x_r\}) = a(b_h, \max\{n, x_{\max}\}) \overset{(viii)}{>} a(b_h - 2, 2 \cdot \max\{n, x_{\max}\}) \geq a(b_h - 2, n + x_{\max}) \geq h(n, x)$

If we use these limits, for a given derivation of a primitive recursive function $f()$, we can compute some value b_f so that $f(x)$ is bounded by $a(b_f, x_{\max})$.

Finally we can use a diagonal argument: If the Ackermann function $a(n, m)$ was primitive recursive, then also $a'(n) = a(n, n)$. So there is some $b_{a'}$ so that $a(b_{a'}, n') \geq a'(n') = a(n', n')$ for all $n' \in \mathbb{N}$. But for $n' > b_{a'}$ this is a contradiction with (vii).

■

Maybe this proof also helps to understand the structure of the Ackermann function a bit. The following exercise might help too:

22 Exercise *Define $b_n(m) := a(n,m)$ for all $n, m \in \mathbb{N}$.*
For which n is $b_n()$ primitive recursive and for which is it not?
Of course you have to prove that your answers are correct.

After Theorem 23, the following theorem is surely a surprising one:

24 Theorem *The set $\{(l,n,m) \in \mathbb{N}^3 | l = a(n,m)\}$ is primitive recursive.*

Unfortunately we can't prove each interesting and/or important theorem in detail. So we will just give a **very rough sketch** here. The sketch is hopefully convincing and clear enough for you to be able to fill in the details if you invest some serious amount of time and energy. In fact we will later on see (and prove) theorems that shed some more light on this theorem and make it look less surprising. Furthermore these theorems give a hint for an approach to an easier proof.
Rough sketch of a proof – just the idea:
You code each computation of $a(n,m)$ – that is: the series of recursion steps following each other – in vectors that are mapped onto each other by a function $f()$: $a(5,4) \to (5,4)$ and $a(5,4) = a(4, a(5,3))$, so $f(5,4) = (4,5,3)$, $f(4,5,3) = (4,4,5,2)$ etc. So you always look at the last two entries and make 3, 2 or 1 entries out of it – depending on whether one of the entries is 0 and which entry it is. These vectors must be coded as a number in $\mathbb{N}$ in a way that makes it possible to determine whether a given number codes a vector corresponding to a computation of some $a(n,m)$. The value of $a(n,m)$ can then be determined by iterating this until just one element remains in the vector.
This sounds like a way to compute $a(n,m)$ – so there must be a problem. This problem is to implement the stop condition. You must use the bounded μ-operator, but that needs an upper bound – and you need the l in the tuple (l,n,m) to compute an upper bound for the bounded μ-operator. The problem when just wanting to compute the Ackermann function $a(n,m)$ is that you don't have this bound and as already the Ackermann function grows faster than any primitive recursive function, no primitive recursive function can give an upper bound on it.

23 Exercise *Give the series of vectors you get when applying this technique to the initial vector $(2,2)$. It starts $(2,2); (1,2,1); \ldots$.*

24 Exercise *Give a reasoning that the kind of induction used in the proof of Theorem 23 works correctly.*
One way would be to rewrite the part of the proof that uses this argument and use two nested normal inductions, but another valid reasoning is also good.

2.3 μ-recursive functions

We surely agree that the Ackermann function should be considered as a computable function, so our approach via primitive recursive functions is not sufficient – the set $\mathcal{P}$ is obviously too small and in order to have some chance to find a definition of computability that reflects our intuition, we have to extend it.
We have already seen (Observation 21) that it won't help to just add the Ackermann function as a base function or the form of recursion used by the Ackermann function as a rule. We would still have functions that a normal computer can *in principle* compute and that would not be covered by the definition.
Another approach is the following:
The parameter l in the previous sketch is only used to compute an upper bound for the bounded μ-operator. If the μ-operator would not need this bound, one could compute the Ackermann function . So this is an approach one could try.
In the future we also don't require the function $f : \mathbb{N}^r \to \mathbb{N}$ to be defined for all $x \in \mathbb{N}^r$. We also allow functions that are defined on just a subset of $\mathbb{N}^r$. Such functions are called partial functions. They can be defined on an arbitrary subset of $\mathbb{N}^r$ – so also on the whole of $\mathbb{N}^r$ or the empty set.

25 Definition *Let $f : \mathbb{N}^{r+1} \to \mathbb{N}$ be a partial function. Then we define $\mu f : \mathbb{N}^r \to \mathbb{N}$ as*

$$\mu f(x) := \begin{cases} min\{m | f(m,x) = 0 \quad \text{and for all } n < m & \text{if the set} \\ \qquad f(n,x) \text{ is defined } \} & \text{is not empty} \\ \text{undefined} & \text{else} \end{cases}$$

Now we will extend the class of primitive recursive functions by adding the μ-operator and requiring that for each function also the function obtained by applying the μ-operator to it is contained. But as we also allow partial functions, we must be careful about where the functions are defined and therefore also have to rewrite the part a bit that we already know from primitive recursive functions.

26 Definition *The class $\underline{\mathcal{R}}$ of μ-recursive functions is the smallest class (that is again the intersection of all such classes) that contains the following base functions and is closed under the rules how to form new functions given in a.) b.) en c.).*

1.) *For all $r, s \in \mathbb{N}$ the constant function*

$$c_s^r : \mathbb{N}^r \to \mathbb{N}, \qquad c_s^r(x) = s \quad \forall x \in \mathbb{N}^r$$

2.) *The successor function*

$$n : \mathbb{N} \to \mathbb{N}, \qquad o(x) = x + 1 \quad \forall x \in \mathbb{N}$$

3.) *For all $i, r \in \mathbb{N}, 1 \leq i \leq r$ the projection on the i-th component – that is the function*

$$p_i^r : \mathbb{N}^r \to \mathbb{N}, \qquad p_i^r(x_1, \ldots, x_r) = x_i \quad \forall (x_1, \ldots, x_r) \in \mathbb{N}^r$$

a.) *Let $r, m \in \mathbb{N}$. If $f : \mathbb{N}^r \to \mathbb{N}$ and $g_1, \ldots, g_r : \mathbb{N}^m \to \mathbb{N}$ are in $\mathcal{R}$, then also $h : \mathbb{N}^m \to \mathbb{N}$ defined as $h(x) := f(g_1(x), \ldots, g_r(x)) \quad \forall x \in \mathbb{N}^m$*

The value $h(x)$ is defined if and only if $g_1, \ldots, g_r$ are defined for x and f is defined for $(g_1(x), \ldots, g_r(x))$. Otherwise $h(x)$ is undefined.

b.) *Let $r \in \mathbb{N}$. If $f : \mathbb{N}^r \to \mathbb{N}$ and $g : \mathbb{N}^{r+2} \to \mathbb{N}$ are in $\mathcal{R}$, then also the (unique) function $h : \mathbb{N}^{r+1} \to \mathbb{N}$ with the properties*

b1.) $h(0, x) = f(x) \quad \forall x \in \mathbb{N}^r$

b2.) $h(n + 1, x) = g(n, h(n, x), x) \quad \forall n \in \mathbb{N}, x \in \mathbb{N}^r$

The value $h(0, x)$ is defined if and only if $f(x)$ is defined. The value $h(n + 1, x)$ is defined if and only if $h(n, x)$ is defined and g is defined for the parameters $(n, h(n, x), x)$. Otherwise the values are undefined. Note that this definition of when a value is defined is recursive.

c.) *If $f : \mathbb{N}^r \to \mathbb{N}$ is in $\mathcal{R}$, then also μf*

27 Theorem *The Ackermann function is μ-recursive.*

Proof: $a(n, m) = \mu(1 \dot{-} \chi_M)(n, m)$, with M the set from Theorem 24.

■

Though it is very vague to argue with our *normal* computer, it can still be used as a guideline to judge some decisions:
It is again intuitively clear that μ-recursive functions can be computed by a *normal* (infinitely large and arbitrarily long running…) computer. Undefined can be seen as a computer that doesn't stop its computation started on that value.
Saying that the functions are undefined for some value is quite different to interpreting the functions as functions $f : \mathbb{N}^r \to (\mathbb{N} \cup \{\text{undefined}\})$. With this second interpretation the computer would have to determine that the value of a function is *undefined* in finite time – which is something else and would in fact be the same class of functions as those $f : \mathbb{N}^r \to \mathbb{N}$ (up to trivial transformations).
This is also what makes the function (hopefully) acceptable: a computer can look for the first zero of a function that is implemented on it and test one number after the other. But if we would not require in the definition of μ-operator that the function $f()$ is defined for all $n < m$, a computer could not do it (at least it is not clear how), as it would never get past the undefined value when trying one value after the other. So without this restriction in the definition it would not be clear that we want to consider all functions in $\mathcal{R}$ as computable.
But with the definition as it is, the set $\mathcal{R}$ contains surely only functions we would like to consider as computable. Whether it is a definition that really matches our intuition or whether we can again find functions we would intuitively call computable that are not contained is not clear yet.

Attention: One of the next two exercises is a trap. In fact the result you are asked to prove is not correct. Find out which one it is and in that case show that the result is not correct.

25 Exercise *Prove: there exists a surjective mapping* $\mathbb{N} \to \mathcal{P}_1$, $n \to f_n$ *with* $\mathcal{P}_1$ *the set of primitive recursive functions* $\mathbb{N} \to \mathbb{N}$ *and a primitive recursive function* $u : \mathbb{N}^2 \to \mathbb{N}$, *so that for all* $n, m \in \mathbb{N}$ *we have* $u(n, m) = f_n(m)$.
Such a function is called universal *as it can simulate all primitive recursive functions* $\mathbb{N} \to \mathbb{N}$. *Later on we will also see a universal Turing machine – a kind of computer with a fixed program that can simulate all other computers with a fixed program.*

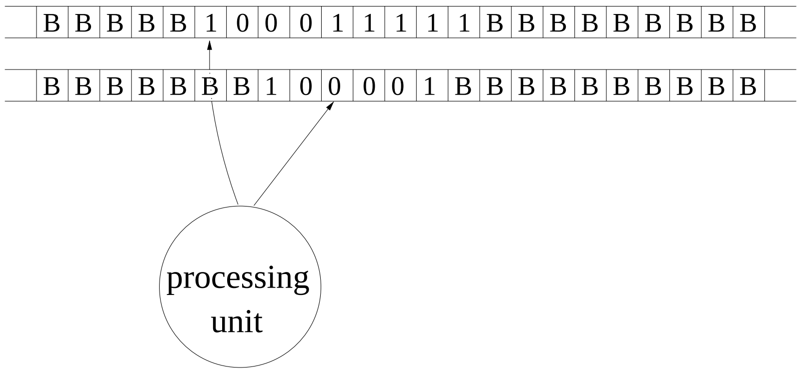
26 Exercise *Prove that a partial function* $f : \mathbb{N}^r \to \mathbb{N}$ *can be written as* $f(x) = \mu g(x)$ *with* $g : \mathbb{N}^{r+1} \to \mathbb{N}$ *a primitive recursive function, if and only if* $M = \{(n, x) | f(x) = n\}$ *is primitive recursive.*

3 Turing machines

In his 1936 article *"On computable numbers with an application to the Entscheidungsproblem"* A.M. Turing described some properties that he finds essential to describe a person doing computations. One of these properties was that a piece of paper was used to do the computations, remember intermediate results, etc. He describes the fact that such a paper is two-dimensional as inessential – a paper where you can write the letters and numbers only next to each other would (according to him) do the same job. In order to be able to write the letters neatly next to each other, he assumed the paper to be partitioned into boxes or cells aligned in a one-dimensional row. In each box only one letter may be written. Furthermore he assumes a finite alphabet, as combinations of letters can be used to represent an infinite variety of symbols. The person doing the computation can see only a finite part of the paper – this can be described as one cell – and for a given content of a cell, there are only finitely many ways to continue. The reason for this is what he calls a finite number of *states of mind*. You can find this article on the internet and a part of it is also reproduced in *D.E. Cohen, Computability and Logic, p.80.*
Based on these observations and thoughts about how people do computations, Turing developed a model of a **theoretical** computing machine. In this approach exactly those functions are called computable that can be computed by such a machine that was later called a Turing machine.
While our first approach was based on which functions and which ways to combine functions we consider to be computable, Turing starts from the **procedure of computing** itself.

3.1 The Turing machine

We will start with an informal description of a Turing machine. One can also say that we start with a description of a physical model of our theoretical machine.


B B B B B 1 0 0 0 1 1 1 1 1 B B B B B B B B B
B B B B B B B 1 0 0 0 0 1 B B B B B B B B B B
processing unit

A Turing machine has k one-dimensional tapes for some $k \in \mathbb{N}$. Each tape has infinitely many cells in each direction. Each of these cells contains exactly one letter from a finite alphabet. For alphabets of Turing machines we always assume that they contains at least the symbols $0, 1, \#, B$. In fact it would be sufficient to require only the symbols $0, 1$ (there will also be an exercise about this), but having also $\#$ and B will make some things a lot easier.
We will always interpret the letter B as *blank* and the letter $\#$ as a sign for separating different parts. The content of a tape is a word (that is a sequence of successive cells) $w_1 \dots w_n$ that contains all letters that are not B and is as short as possible.
On each of the tapes you have a read/write head that stands on exactly one of the cells. The head can read the sign written in that cell and replace it by another letter from the alphabet. The heads can move independent of each other on each tape. They can move one cell right or left or keep their position.
The heads are connected with some central processing unit (let's call it CPU). Based on the tuple of letters that are read, this CPU decides which letters are written by the heads and in which direction the heads move.
The CPU has a finite number of states. In each step it can freely change between the states, but the next action of the heads and the next state of the CPU depend only on the tuple of letters read by the heads and the state the CPU is in at that moment. Saying that it depends on the *tuples* of letters read is to emphasise that it can make a difference whether you read a 1 on tape 1 and a B on tape 2 or the other way around.
There are at least two special states of the CPU: exactly one start state and at least one halting state. The machine always starts in the start state and stops when a halting state is reached. When a halting state is reached, no more modifications of positions, contents or state take place.

A computation of a Turing machine started on given contents of the tapes is starting the machine with the heads of the first (leftmost) letter of the content of each tape, together with all movements of the heads, all modified contents of the tapes, etc. until a halting state is reached. Note that it is possible that this halting state is never reached. In this case we call it an infinite computation. For an **exact** definition of a computation, see Definition 30. If the machine stops, the contents of the tapes when the machine is in a halting state is the result of the computation. If the machine does not stop, there is no result, so the result is undefined.
Now the exact mathematical definition of a Turing machine:

28 Definition *A* k-tape Turing machine *is a 5-tuple*

$$M = (Z, A, \delta, z_0, E)$$

with the following properties:

- *Z is a finite set – the set of states.*
- *A is a finite set with the properties $\{0, 1, \#, B\} \subseteq A$, $Z \cap A = \emptyset$. This is the alphabet.*
- *δ is a function $Z \times A^k \to Z \times A^k \times \{L, R, N\}^k$. We call this the transition function. We do not require δ to be defined for all elements of $Z \times A^k$. L, N, R describe the movements of the heads* left, neutral, right. *For $1 \leq i \leq k$ the i-th component of the tuple $\{L, R, N\}^k$ describes how the head moves on tape i.*
- *$z_0 \in Z$ is the start state.*
- *$E \subseteq Z$ is the set of halting states. We always choose E as the set of all states z so that $\delta(z, w)$ is not defined for any $w \in A^k$.*

We did not require δ to be a total function, but if we did, it would not change much for the following arguments.

A bit notation:

- If A is an alphabet, then let $A_- := A \setminus \{B\}$.
- A^* is the set of all finite sequences of letters from A. A^+ is the set of all sequences in A^* except for the empty one.
- If $w \in A^*$ we write $|w|$ for the length of w (that is: the number of letters in w) and w_i for the i-th letter.

Definition and Remark
The entire state of a Turing machine at a certain time (that is: after a certain step) can be described by the state the CPU is in, the contents of the tapes and the positions of the heads on the tapes. If the content is finite, this state can be described by a k-tuple of finite strings with the i-th element of the form $u^{(i)}zw^{(i)}$ with $z \in Z$, $u^{(i)}, w^{(i)} \in A^*$. In this string, z is the state of the machine and $u^{(i)}$ and $w^{(i)}$ are chosen in a way that on tape i left of $u^{(i)}$ and right of $w^{(i)}$ there are only blanks and that the head is on the first letter of $w^{(i)}$ or on the first blank to the right in case $w^{(i)}$ is empty.
We call such a vector a configuration of a Turing machine. If $z \in E$ we call it an end configuration.

Now we will give an exact definition of *a computation of a Turing machine.*

29 Definition *Let c, c' be configurations of a k-tape Turing machine and $\bar{a} = (\bar{a}_1, \ldots, \bar{a}_k)$ the k-tuple of letters that are read by the heads in configuration c. Let z be the state in configuration c and $\delta(z, \bar{a}) =: (z', \bar{a}', \bar{r})$, $\bar{a}' \in A^k$, $\bar{r} \in \{R, L, N\}^k$.*
Then c' is called the successor configuration of c (we write $c \vdash c'$) if and only if we have for all $c_i, c'_i, 1 \leq i \leq k$:

- *If $c_i = ubz\bar{a}_i v$ with $u, v \in A^*$, $b \in A$ (so there are letters to the left and to the right of the position of the state in the i-th component of the configuration), then*
  - $c'_i = uz'b\bar{a}'_i v$ *if* $\bar{r}_i = L$
  - $c'_i = ubz'\bar{a}'_i v$ *if* $\bar{r}_i = N$
  - $c'_i = ub\bar{a}'_i z'v$ *if* $\bar{r}_i = R$
- *If $c_i = z\bar{a}_i v$ with $v \in A^*$, (so there are only letters to the right), then*
  - $c'_i = z'B\bar{a}'_i v$ *if* $\bar{r}_i = L$
  - $c'_i = z'\bar{a}'_i v$ *if* $\bar{r}_i = N$
  - $c'_i = \bar{a}'_i z'v$ *if* $\bar{r}_i = R$
- *If $c_i = ubz$ with $u \in A^*$, $b \in A$, (so there are only letters to the left), then*
  - $c'_i = uz'b\bar{a}'_i$ *if* $\bar{r}_i = L$
  - $c'_i = ubz'\bar{a}'_i$ *if* $\bar{r}_i = N$

  - $c'_i = ub\bar{a}'_i z'$ *if* $\bar{r}_i = R$

- *If* $c_i = z$ *(so there are no letters at all), then*
  - $c'_i = z'B\bar{a}'_i$ *if* $\bar{r}_i = L$
  - $c'_i = z'\bar{a}'_i$ *if* $\bar{r}_i = N$
  - $c'_i = \bar{a}'_i z'$ *if* $\bar{r}_i = R$

If $\delta(z, \bar{a})$ *is undefined, there does not exist a successor configuration – it is also undefined.*
Attention: *If you add just blanks to the beginning or end of a configuration, you describe the same state of the Turing machine– but for the definition of the successor configuration it makes a difference whether you do it or not!*

Agreement:
From now on, we assume that when a Turing machine is started, on each tape the heads are on the first (that is: leftmost) letter of the content that is not a blank (if such a letter exists). If the tape is empty – that is: contains only blanks – it does not matter where the head is.

30 Definition *Let* $M = (Z, A, \delta, z_0, E)$ *be a* k*-tape Turing machine.*
A finite sequence $c_0, \ldots, c_t$ *of configurations is called a finite* computation *of the Turing machine* M *started on* $w = (w_1, \ldots, w_k) \in (A^*)^k$, *if and only if for* $1 \leq i \leq k$ *the string* w_i *is empty or starts and ends with a letter from* A_- *and*

i.) c_0 *has the form* $(B \ldots Bz_0w_1B \ldots B, \ldots, B \ldots Bz_0w_kB \ldots B)$

ii.) $c_i \vdash c_{i+1}$ *for* $0 \leq i \leq t-1$

iii.) c_t *is an end configuration.*

An infinite sequence of configurations that fulfils only i.) and ii.) (ii.) of course for all $i \in \mathbb{N}$*), is called an infinite computation of* M *started on* w.
In cases where for one of the configurations no successor configuration is defined (that is: δ *is not defined for the values given by the configuration), we also call it an infinite computation.*
Attention: *There are infinitely many sequences describing the same computation of the physical Turing machine. These different theoretical computations differ in the number of blanks at the very left or very right of the strings in the start configurations (with all the consequences for the later configurations). The computation with a start configuration of the form* $(z_0w_1, \ldots, z_0w_k)$ *is called the* canonical computation *of* M *started on* $w_1, \ldots, w_k$.

4 Example • *As an easy first example, we will give a 1-tape Turing machine $M_{1,n}$, that when started on an empty tape writes the binary representation of the number $n \in \mathbb{N}$ on the tape and stops on the beginning of the number:*

Let $b_1 \dots b_k$ be the binary representation of n with b_1 the highest valued bit.

- $Z := \{z_1, \dots, z_k = z_0, z_e, z_{e'}\}$
- $A := \{0, 1, \#, B\}$
- $E := \{z_e, z_{e'}\}$
- *δ is defined as follows:*
  - $\delta(z_i, B) = (z_{i-1}, b_i, L)$ *for* $2 \leq i \leq k$
    $\delta(z_i, a) = (z_{e'}, b_i, L)$ *for all* $a \in A_-, 2 \leq i \leq k$
  - $\delta(z_1, B) = (z_e, b_1, N)$
    $\delta(z_1, a) = (z_{e'}, b_1, N)$ *for all* $a \in A_-$

The state $z_{e'}$ is used here to remember that a non-blank letter was overwritten. This is not necessary to fulfil the specifications of this machine. It is an example to show what different end states can be useful for. In this case stopping in z_e means: number written – no non-blank letter overwritten. *Stopping in z'_e means:* stopped as I came across non-blank letters. *So it can be used in a similar way as throwing an exception in Java.*

- *A bit more complicated: a 1-tape Turing machine $M_2 =: M_{+1}$, that started on a content that is the binary representation of a number n, limited by some # or a blank, modifies it so that in the end configuration the binary representation of $n+1$ is on the tape, limited to the right by the same letter as before and the head is on the first letter of the binary representation.*
  - $Z := \{z_r = z_0, z_a, z_l, z_e\}$
  - $A := \{0, 1, \#, B\}$
  - $E := \{z_e\}$
  - *δ is defined as follows:*
    - *z_r looks for the end of the binary representation.*
      $\delta(z_r, 0) = (z_r, 0, R)$
      $\delta(z_r, 1) = (z_r, 1, R)$
      $\delta(z_r, \#) = (z_a, \#, L)$
      $\delta(z_r, B) = (z_a, B, L)$

  * z_a *adds* 1*: a* 1 *becomes a* 0 *until a blank or a* 0 *is found. Then a* 1 *is written and the state of the CPU is changed.*
    $\delta(z_a, 0) = (z_l, 1, L)$
    $\delta(z_a, 1) = (z_a, 0, L)$
    $\delta(z_a, \#) = (z_e, 1, N)$
    $\delta(z_a, B) = (z_e, 1, N)$

  * z_l *looks for the left end of the number.*
    $\delta(z_l, 0) = (z_l, 0, L)$
    $\delta(z_l, 1) = (z_l, 1, L)$
    $\delta(z_l, \#) = (z_e, \#, R)$
    $\delta(z_l, B) = (z_e, B, R)$

If this machine is not started on the left end of one number n*, but for example on the beginning of a second of two numbers separated by a* #*, then the separating* # *may be removed and the number on the tape is probably everything else than* $n + 1$ *– no matter whether* n *was the first or second number. We do not claim anything for inputs that do not fulfil the specifications mentioned.*

- *If the content on tape 1 consists of letters from* $A_{\_}$ *and all other tapes are empty, we call that a* <u>*regular content*</u> *of a Turing machine.*

  The term <u>*regular content*</u> *of a tape denotes a content from* $A_{\_}^*$ *or an empty content.*

  A 2-tape Turing machine M_3*, that – started on a regular content – copies the content of tape 1 onto tape 2, but in reverse order and also clears tape 1 and stops on the first non-blank letter on tape 2 can be defined as follows:*

  - $Z := \{z_0, z_e\}$
  - $A := \{0, 1, a, \#, B\}$
  - $E := \{z_e\}$
  - δ *is defined as follows:*
    * $\delta(z_0, B, y) = (z_e, B, B, N, R)$ *for all* $y \in A$
      $\delta(z_0, x, y) = (z_0, B, x, R, L)$ *for all* $x \in A \setminus \{B\}$, $y \in A$

Attention: *We don't claim anything about what happens when the input is not according to the specifications. Then* **everything** *can/may happen...*

5 Example *How do computations look, when we apply these example machines?*

- *The machine $M_{1,5}$ started on an empty tape, makes the following computation (the binary representation of* 5 *is* 101*):*

  $z_3, z_2B1, z_1B01, z_e101$

- *The machine M_2 started on a tape with content* 101 *makes the following computation:*

  $z_r101, 1z_r01, 10z_r1, 101z_r, 10z_a1B, 1z_a00B, z_l110B, z_lB110B, Bz_e110B$

- *The machine M_3 started on the content $01aa$ on tape* 1 *and an empty tape 2 makes the following computation:*

  $(z_001aa, z_0), (Bz_01aa, z_0B0), (BBz_0aa, z_0B10), (BBBz_0a, z_0Ba10),$
  $(BBBBz_0, z_0Baa10), (BBBBz_eB, Bz_eaa10)$

31 Definition *If $c_0, \ldots, c_t$ is a finite computation of a k-tape Turing machine, then t is called the length,time or time complexity of the computation. If the computation is canonical, then $\sum_{j=1}^{k}(\max_{0\leq i\leq t}\{|(c_i)_j|-1\})$ is called the memory complexity or tape complexity of the computation. As configurations grow monotonically, the* max *is only for better understanding. In fact the maximum is always attained for $i = t$. If the movement when going to the end state is neutral, the memory complexity is (almost always) the number of (different) cells that belonged to the original content or were written on by the head during the computation – even if what was written was the same as what stood there before. If the last move is not neutral but going left and the head visits a new blank cell, the definition counts one cell more. If we applied the formula to a non-canonical computation, this interpretation would of course not work!*
If the memory complexity of an infinite computation $c_0, c_1, c_2, \ldots$ is defined as the limit $t \to \infty$ of the memory complexities of the finite partial computations $c_0, c_1, \ldots, c_t$, the memory complexity of the infinite computation can still be finite.

If $f : \mathbb{N} \to \mathbb{N}$ is a (possibly partial) function, then a Turing machine M is called time bounded by $f(n)$ if for all n for which $f(n)$ is defined and for all possible $w = (w_1, \ldots, w_k)$ with $|w| = |w_1|+\cdots+|w_k| = n$, we have that there is a finite computation of M started on w (including that all changes of state are well defined) and that the length of the computation is at most $f(n)$. The length of a computation does not depend on whether the computation is canonical or not.

M is called memory bounded or tape bounded by $f(n)$ if for all n for which $f(n)$ is defined and for all possible w with $|w| = n$, we have that a computation of M started on w is defined and that the tape complexity of a **canonical** *computation of M started on w is at most $f(n)$.*
Exercise: *What would be the consequences if we would not require the computation to be canonical?*
So we make a worst case approximation. For a given length n the input with length n for which the most time, resp. tape is needed, determines the bound.

6 Example
- *In this single case we will use m for the length of the input in order not to mix it up with the n in* the machine $M_{1,n}$,

  The 1-tape Turing machine $M_{1,n}$ is time bounded by the function $t(0) = k$ and $t(m) = 1$ for $m > 0$ – with k the number of bits in the binary representation of n. It is tape bounded by $b(m) = \max\{m+1, k\}$. Note that for the limits you also have to take cases into account that don't meet the specifications of the machine, where you may e.g. require an empty tape or some other specialised input. Also in those cases a computation takes place!

- *The machine M_2 is time bounded by the function $t(n) = 2n + 2$ and tape bounded by the function $b(n) = n + 2$.*

- *The machine M_3 is time bounded by the function $t(n) = n + 1$ and tape bounded by the function $b(n) = 2n + 2$.*

32 Definition *A Turing machine is called regular, if each finite computation that starts with a configuration where on each nonempty tape the content is regular and the head is on the first (leftmost) letter of the content also ends with such a configuration.*

27 Exercise *Give a detailed description of a regular 1-tape Turing machine $M_{i,k}$ that started on a tape with $k \geq i \geq 1$ binary representations of numbers $n_1, \ldots, n_k$ separated by # (so the input in that case is $n_1\#n_2\#\ldots\#n_k$) removes all the separating letters # and all numbers except the i-th.*
Give functions that bound time and tape complexity in the worst case. Choose these functions so that you have sharp bounds.

28 Exercise *Give a detailed description of a regular 1-tape Turing machine M_{-1} that started on a tape with only the binary representation of a number n modifies the content to the binary representation of $n \dot{-} 1$. A binary representation must have no leading zeros.*

Give functions that bound time and tape complexity in the worst case. Choose these functions so that you have sharp bounds.

29 Exercise *Give a detailed description of a* 1*-tape Turing machine that stops if and only if the tape contains a letter different from B. For this exercise you may of course not assume that the machine starts on the first letter that is not a B if the tape is not empty. . .*

30 Exercise *For each of the following statements you should give a proof or show that it is wrong:*

- *There is a total function $g : \mathbb{N} \to \mathbb{N}$ with $\lim_{n\to\infty} g(n) = \infty$ so that for each Turing machine M and each $t \in \mathbb{N}$ we have:*

  a.) *If there is a computation of M for which the tape complexity of the partial computation with t steps is strictly less than $g(t)$, then it is an infinite computation.*

- *For each Turing machine M there is a total function $g : \mathbb{N} \to \mathbb{N}$ with $\lim_{n\to\infty} g(n) = \infty$ so that a.) is valid for each t.*

To understand how the alphabet and the set of states can *work together* and to get a feeling for the construction of Turing machines, we will give an example how the states can remember words of fixed lengths. In the example the length will be just 1 to keep it simple.

7 Example *The following* 1*-tape Turing machine $M_{\text{move right}}$ moves some regular content of a tape one cell to the right and stops on the first letter of the content:*

- $Z := \{z_0, z_e, z_l\} \cup \{z_{(a)} | a \in A, a \neq B\}$
- *Let A be an arbitrary alphabet containing $\{0, 1, \#, B\}$.*
- $E := \{z_e\}$
- *δ is defined as*
  - $\delta(z_0, a) = (z_{(a)}, B, R)$ *for all* $a \in A, a \neq B$
    $\delta(z_0, B) = (z_e, B, N)$

- *$z_{(a)}$ remembers that one cell to the left there was the letter „a“:*
  $\delta(z_{(a)}, b) = (z_{(b)}, a, R)$ *for all* $b \in A, b \neq B$
  $\delta(z_{(a)}, B) = (z_l, a, L)$

- *z_l searches the left end of the content:*
  $\delta(z_l, a) = (z_l, a, L)$ *for all* $a \in A, a \neq B$
  $\delta(z_l, B) = (z_e, B, R)$

Even though it is a simple example, one can still see that for arbitrary (fixed) lengths there are Turing machines that can store words of this length in the states and this way make the decision on the next step in fact depending on more than just the letter just read. **But:** *for each given Turing machine the length of the words to store is bounded by a constant. For longer words you need other (and larger) Turing machines!*

3.2 Combinations and modifications of Turing machines

We have already seen that it is quite an effort to explicitly write up even simple Turing machines with all their states. To avoid this, we will now write up Turing machines by combining them from smaller and already known Turing machines.

Let $M_i = (Z_i, A_i, \delta_i, (z_0)_i, E_i)$, $1 \leq i \leq n$ be k-tape Turing machines.
As renaming states does not change the way a machine works, we assume – just to be able to write up things more easily – that the sets of states of all machines that we use are pairwise disjoint. If that was not the case, we could simply rename them.
We will now combine all these machines to a new machine $M_\cup := (Z, A, \delta, z_0, E)$:

- $Z := \dot{\bigcup}_{i=1...n} Z_i$
- $A = \bigcup_{i=1...n} A_i$
- for all i and $z \in Z_i$, $z \notin E_i$: $\delta(z, \bar{a}) := \delta_i(z, \bar{a})$ for all $\bar{a} \in A_i^k$. For $\bar{a} \notin A_i^k$ one can define $\delta(z, \bar{a})$ arbitrarily (e.g. stop in some special new end state).

What we have not defined so far, are z_0, E and the function δ for stop states of the partial machines that we do not want to be in E.
These definitions will determine the way the partial machines *work together*.
One machine must start the computation. If that is M_{i_0}, we define:

- $z_0 := (z_0)_{i_0}$

If the computation must continue with machine $M_{j(q)}$ after for $1 \leq i \leq n$ machine M_i has stopped its computation and ended in stop state q, we define:

- $\delta(q, \bar{a}) := ((z_0)_{j(q)}, \bar{a}, N, \ldots, N)$ for all $\bar{a} \in A^k$

Note that the next machine can depend on the stop state of a given machine and not only on the machine that just finished its computation. As we assume all states to be different, it depends in fact only on the stop state...
Finally we define E to be the set of states from $\bigcup_{i=1\ldots n} E_i$ after which no other machine must take over the computations.
A good way to represent such a machine is in a flow chart with arrows describing the changes between the stop state of one machine to the start state of another. Arrows leading to **STOP** belong to states in E.
The following figure 1 shows an example for machines $M_1, \ldots, M_7$ and $\bigcup_{i=1\ldots 7} E_i = \{o, p, q, r, s, t, u, v, w, x, y, z\}$.

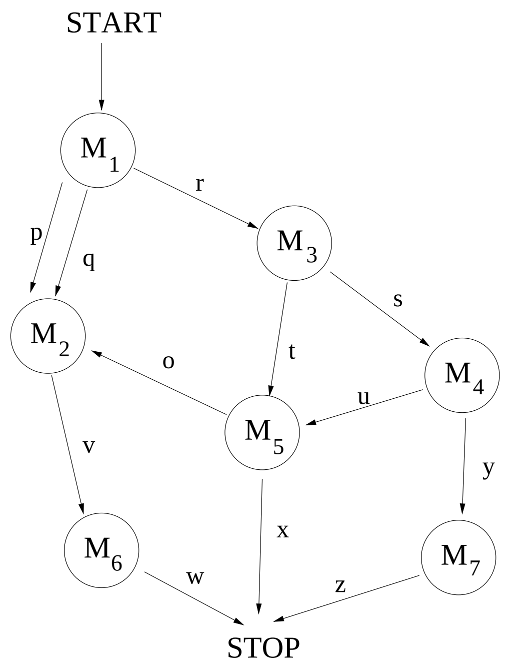


Figure 1:

If the flow chart is linear – meaning also that from all stop states of a certain machine the computation switches to the same next machine – we don't have to mention all stop states explicitly and can just write
$START \to M_1 \to M_2 \to \cdots \to M_k \to STOP$
or even
$M_1 \to M_2 \to \cdots \to M_k$

8 Example *The* 1*-tape Turing machine* M_{remove} *is defined as*

- $Z := \{z_0, z_e\}$
- *Let* A *be an arbitrary alphabet containing* $\{0, 1, \#, B\}$.
- $E := \{z_e\}$
- δ *is defined as*
  - $\delta(z_0, a) = (z_0, B, R)$ *for all* $a \in A, a \neq B$
    $\delta(z_0, B) = (z_e, B, N)$

This machine removes a regular content of a tape.
Together with the example machine $M_{1,s}$ *we already saw, we can now build a machine* $M_{c,s}$ *that started on a regular content of the form* $n_1\#n_2\#\ldots\#n_r$, *with* n_i *binary representations of natural numbers, removes this content and writes the constant* s *on the tape. This machine – in a certain sense – implements the constant primitive recursive function* c_s^r.
This machine $M_{c,s}$ *is given by* $\quad M_{remove} \to M_{1,s}$.

We can also modify a given k-tape Turing machine $M = (Z, A, \delta, z_0, E)$ for a given $m \geq k$ and pairwise different numbers $1 \leq i_1, i_2, \ldots, i_k \leq m$ so that we get an m-tape Turing machine $M(i_1, \ldots, i_k) = (Z', A', \delta', z_0', E')$ that – for $1 \leq j \leq k$ – works on tape i_j as M works on tape j and does not modify the other tapes:

- $A' := A$
- $Z' := Z$
- $z_0' := z_0$
- $E' := E$

- if $\delta(z,\bar{a}) = (z', \bar{b}, \bar{d})$ with $\bar{a}, \bar{b} \in A^k$, $\bar{d} \in \{R,N,L\}^k$, then the transition function δ' is for $1 \leq j \leq k$ and $\bar{a}' \in A^m$ with $\bar{a}'_{i_j} = \bar{a}_j$ defined as

  $\delta'(z,\bar{a}') = (z', \bar{b}', \bar{d})$ with $\bar{b}' \in A^m$, $\bar{d}' \in \{R,N,L\}^m$ with $\bar{b}'_l = \bar{a}'_l$ and $\bar{d}'_l = N$ if $l \notin \{i_1, \ldots, i_k\}$ and $\bar{b}'_{i_j} = \bar{b}_j$ and $\bar{d}'_{i_j} = \bar{d}_j$ else.

9 Example *Let M_3 be the Turing machine from Example 4. If $M_3(2,5)$ is started with some content on tape 2 and an empty tape 5, the content of tape 2 is written in reversed order on tape 5, tape 2 is cleared and the machine stops with the head on the first letter of the content of tape 5. Tapes different from 2 and 5 are not modified.*

33 Definition *We call a configuration a possible $\underline{\text{transition configuration}}$ if each tape is either empty or the head is on the first letter of some regular content.*

So a Turing machine is $\underline{\text{regular}}$, if each finite computation that starts with a transition configuration also ends with one.
A regular 1-tape Turing machine $M_{=0}$ that started on a regular content decides whether the content is a single 0 and stops in state *yes* if this is the case and in state *no* otherwise is easy to describe.
Assume now that $m, k \in \mathbb{N}$, $m \geq k$ and an arbitrary regular m-tape Turing machine M are given. Then we call the following Turing machine $M_{\text{while tape } k \neq 0}$:

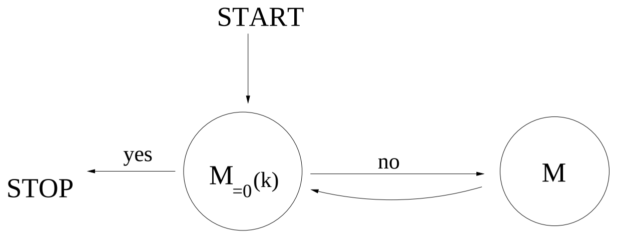


Figure 2: The Turing machine $M_{\text{while tape } k \neq 0}$

10 Example *We will use the machines M_{+1} and M_{-1} we have already constructed.
If M is the Turing machine $M_{+1}(1) \rightarrow M_{-1}(2)$, then the result of $M_{\textit{while tape } 2 \neq 0}$ started on the binary representation of x on tape 1 and the binary representation of y on tape 2 is the binary representation of $x + y$ on tape 1 and 0 on tape 2.*

31 Exercise
- *What is the tape and time complexity of this machine? Giving the bounds in the $O()$-notation is sufficient.*
  Can you design a faster machine?
- *Describe a Turing machine with at least two tapes that computes $x \cdot y$ on tape 1, if it is started with its heads on the first letter of the binary representations of x on tape 1 and y on tape 2 and otherwise empty tapes.*

3.3 Simulation of Turing machines

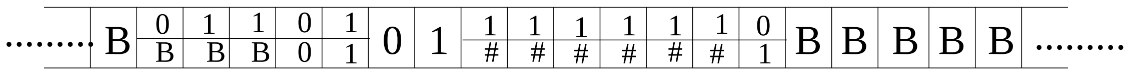
How important is it to allow k tapes instead of just 1. Can machines with 2 tapes compute more functions than those with just 1? Or are they *equally powerful*? And what does *equally powerful* mean? It is of course obvious that a 1-tape Turing machine can not produce a *result* that has some value on tape 1 and one on tape 2, so in a certain sense a machine with more tapes can do something that a machine with 1 tape can't. But is this difference really important?
We will now define what we mean when saying that two machines *can do the same.*

34 Definition *Let M with alphabet A and M' with alphabet $A' \supseteq A$ be Turing machines. We say that M' simulates M if for all inputs $w \in A^*_\_$ (that is for all words that are on the tapes in the beginning) either both machines stop or both don't. If both machines stop, the result of M started on w and that of M' started on w must be the same. That is: All tapes that are present in both machines must have the same content. If one machine has more tapes than the other, then the content of the tapes that are present in one machine, but not the other, must be empty after the computations.*
If $A = A'$ and the number of tapes is also the same, this definition is symmetric in M en M'. So in this case M simulates M' if and only if M' simulates M.

This definition only requires identical results – the way how the results are reached is not important. So also the time and tape bounds can be extremely different – one machine can be very fast and the other very slow!
But although at this moment we are mainly interested in computability, we should still keep an eye on complexity – that is: how efficiently are things computable that are computable. So we will also discuss how efficiently one machine can simulate the other. In most cases when we analyse this, we will use the $O(), o()$ and $\Omega(), \omega()$ notations you already know.

32 Exercise *Give a detailed proof that a $t(n)$ time bounded 1-tape Turing machine M can be simulated by a 1-tape Turing machine M' that is also $t(n)$ time bounded and moves the head in each step.*
We do not require the head of M' to be in the same position after the computation.

For the following proofs, the method of *subdividing tapes in <u>tracks</u>* is very helpful. This method does not use a new type of Turing machine, but is just a way to interpret some Turing machines with large alphabets in a way that makes the argumentation easier. There are **never** real tracks on the (after all not physically existing) tapes!
If A and C are alphabets and if a Turing machine has an alphabet that contains $A \times C$ as a subset (next to other letters like $0, 1, \#, B$), then we can **interpret** letters of the form $(a, b) \in A \times C$ **as if** the tape had two tracks and the first track contains an a and the second a b.
This has a certain similarity with 2 tapes, but as we have in fact just 1 tape, the head can of course not move in different directions on the imaginary tracks.



.... B 0/B 1/B 1/B 0/0 1/1 0 1 1/# 1/# 1/# 1/# 1/# 1/# 0/1 B B B B B ....

If we say that a k-tape machine $M' = (Z', A', \delta', z'_0, E')$ with alphabet $A' \supseteq A \cup (A \times C)$ acts on track 1 of the tapes in the same way as $M = (Z, A, \delta, z_0, E)$ on the whole tapes, this means

- $Z' \supseteq Z$
- $z_0 = z'_0$
- $E' = E$
- for all $z \in Z, a_1, \ldots, a_k \in A$ we have that if $\delta(z, a_1, \ldots, a_k) = (\bar{z}, b_1, \ldots, b_k, d_1, \ldots, d_k)$ then for $a'_i \in (\{a_i\} \cap \{B\}) \cup (\{a_i\} \times C)$, $1 \leq i \leq k$ we have that $\delta'(z, a'_1, \ldots, a'_k) = (\bar{z}, b'_1, \ldots, b'_k, d_1, \ldots, d_k)$, with the property that if $a'_i = a_i = B$ then $b'_i = (b_i, B)$ and if $a'_i = (a_i, y) \in A \times C$ then $b'_i = (b_i, y)$.

Note that we do not fix what the machine must do if a letter from $A \setminus \{B\}$ is read. By the requirement that $Z' \supseteq Z$, machines with disjoint sets of states can never fulfil these requirements, which seems quite a strong restriction. But as we can simply rename states to get another machine that works the same way and fulfils $Z' \supseteq Z$, this restriction is just of a formal nature.

33 Exercise *What do you think is the reason that the letter B is treated differently than e.g. $0, 1, \#$? Why does the definition fix the working of the machine when a B is read?*

A completely analogous interpretation can be given for the case of more than 2 tracks and for the case of using different tracks on different tapes.

35 Lemma *If M is a $t(n) \geq n$ time bounded and $b(n)$ tape bounded k-tape Turing machine, then there is an $O(b(n))$ tape bounded and*

- *$O(t(n) + b^2(n))$ if $k = 1$*
- *$O(t(n))$ if $k \geq 2$*

time bounded k-tape Turing machine M', that simulates M and where the result never uses cells left of the cells used by the input.

Proof: The idea is that one marks the first cell of the input and afterwards moves the output so that it does not start before this mark. The reason for the different time bounds is that this moving of the output is easier if you have more than just one tape.

Setting a mark is done by using tracks: we use 2 tracks, but the second track is only used to store the mark (e.g., an x). On the first track we work exactly like the machine that is to be simulated.

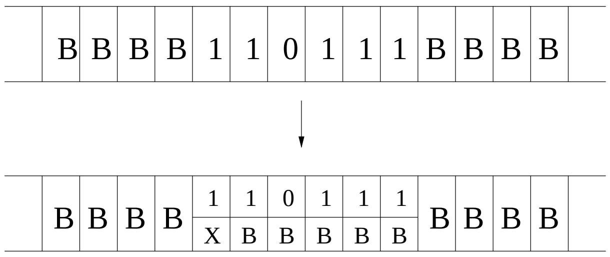
W.l.o.g. we assume that M never prints a B. If M does print a B we just consider a modified machine M_{new} that prints an alternative letter B' when M prints a B and the rules when reading B or B' are identical to the ones when M reads a B. After the run – that is after the computation of the modified machine M_{new}, resp after the computation of the simulating machine – all B' can be replaced by B again to get exactly the same result as M.

Exercise: Can these changes be done within the given time and tape limits?

If A is the alphabet of M and $x \notin A$, then let $A' = A \cup (A \times (\{x\} \cup A))$ be the alphabet of M'.

First we will define useful partial machines:

$M_{\text{mark and make tracks}}$ transforms a regular content $w = w_1 \dots w_n$ with letters from A to $w = (w_1, x)(w_2, B)(w_3, B) \dots (w_n, B)$ and replaces on empty tapes the B where the head is situated by (B, x). At the end $M_{\text{mark and make tracks}}$ returns with the heads to the beginning of the content on each tape. In the meantime we have enough experience with Turing machines to be able to see that this is possible without actually giving all the δ's! **But:** saying that such a machine "obviously exists" means that one should be able to write up the exact details if necessary. If somebody thinks that he can not do that, he should definitely try that as an exercise!



The machine $\bar{M}$ works on the first track exactly the way that M works – except that $\bar{M}$ always stops with the heads on the first non-blank letter of each tape, even if M was not regular. So $\bar{M}$ still has to search for the left end of the content, but as $\bar{M}$ never prints B, this is easy.

Our machine starts with $M_{\text{mark and make tracks}} \to \bar{M}$. Afterwards the content must still be moved and translated back to the alphabet A. The time bounds for this part depends on whether the number k of tapes is 1 or at least 2.

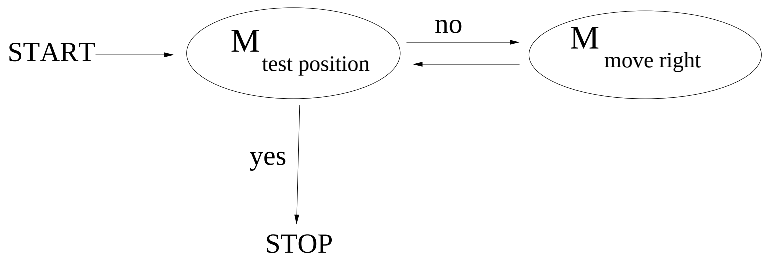
First the more difficult case $k = 1$:

As the machine never prints a B, the content begins in the cell with an x on track 2 or to the left of that cell.

The machine $M_{testposition}$ tests whether the head is in a cell with an x on track 2. If that is the case (the content is moved far enough) the machine switches to stop state *yes*, otherwise to stop state *no*.

The machine $M_{\text{move right}}$ was already discussed in an example. For our application the machine works exactly as in that example – except that only the content of track 1 is moved (in order not to move the x) and afterwards the new (B, B) that is now on the first position is replaced by a B.

The machine M_{move} is then defined as



The machine $M_{\text{remove tracks}}$ translates letters $(a,b) \in (A \times C)$ (back) to $a \in A$. Also in this case it is very useful that we never print a B as this way after the translation the start of what has to be translated is easier to find.

The machine M' is now defined as $M_{\text{mark and make tracks}} \to \bar{M} \to M_{\text{move}} \to M_{\text{remove tracks}}$.

The translations can surely be done in time $O(n)$, resp. $O(b(n))$ with $b(n) \leq 2t(n)$. The simulation needs time $O(t(n))$ and moving right by 1 cell needs $O(b(n))$ steps. As in the worst case the content needs to be moved by 1 cell $O(b(n))$ times, we get a time bound of $O(t(n) + b^2(n))$.

The tape complexity is at most 2 times the complexity of the simulated machine, so it is $O(b(n))$.

In case of more than just 1 tape, one **can** work in the same way as for a 1-tape Turing machine, but moving the content can be done more efficiently – even in time $O(b(n))$ – as the content can be buffered on another tape. The machines M_{Vi}, $2 \leq i \leq k$ work as follows:

It searches the x on track 2 of tape 1 and copies track 1 of tape i to track 2 of tape 1 starting in the cell after the x. At the same time, track 1 of tape i is erased (overwritten by blanks). Then the machine goes back to the mark x on both tapes and copies the content of track 2 of tape 1 beginning in the cell after the x back to track 1 of tape i so that it stands right of the mark. The content right of the x on track 2 of tape 1 is erased during this copy operation. Afterwards both heads go back to the beginning of the content on their tapes. This can clearly be done in time $O(b(n)) = O(t(n))$.

The machine M_{V1} works completely analogously – only with the roles of tape 1 and tape 2 interchanged.

$M_{V1} := M_{V2}(2, 1, 3, \ldots, k)$

M_{move} is in case of more than 1 tape defined by

$M_{V1} \to M_{V2} \to \cdots \to M_{Vk}.$

■

In this case we did not *explicitly* use the definitions of time and tape complexity involving numbers of configurations or the size of configurations, but rather informal arguments like *used cells* or *number of steps*. But when reading this, one must always check whether these arguments can also be expressed using the formal definitions. If this is not the case, there is a serious problem...So *implicitly* we did use the formal definitions and have just used some kind of shorthand notation to make the proofs a bit more readable.

36 Lemma *If M is a $t(n) \geq n$ time bounded and $b(n)$ tape bounded k-tape Turing machine then there is a $O(b(n))$ tape bounded and in case $k \geq 2$ $O(t(n))$ time bounded and in case $k = 1$ $O(t(n) + b^2(n))$ time bounded k-tape Turing machine M^* that simulates M and never visits cells left of the start position of the heads.*

Proof: We have now enough experience with Turing machines that it is not necessary to work out all details of the proofs in order to be convinced that they work. Sometimes it is sufficient to give the most important ideas and you could fill in the details yourself to make it a really nice and complete (but much longer) proof.

Here we will also just give the basic idea:

First we assume that the result is never in cells left of the original position of the heads. If this is not the case for M, we just switch to the machine M' from Lemma 35. Of course we have to take into account that M' already has larger time and tape complexities.

We construct M^* – expressed a bit pictorially – by *bending* the tape as shown in Figure 3.

The tapes right of the start positions of the head are again subdivided in 2 tracks and an x is placed to mark the start position.

Each state is replaced by 2^k states (note that this is a constant) so that for each tape the state can code whether to work on track 1 or 2. Then the Turing machine works just like M' on track 1 resp. 2, but when you work on track 1, you move like M' while on track 2 right and left moves are interchanged.

A special case is when you are on the cell with the x: on this cell you have to change between the states for track 1 and those for track 2 if you work on track 1 and go left or reached the cell from track 2.

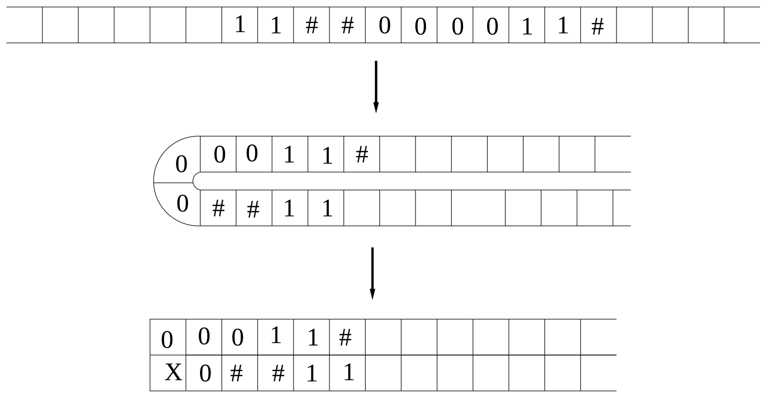


Figure 3: *Bending* a tape.

Afterwards the tracks have to be removed, but as the result will be on track 1, nothing has to be moved.

With all the experience from the exercises and the earlier proofs you can surely fill in the details of this proof. . .

■

In some books Turing machines are defined in a way that the tape is only infinite in one direction. This looks more restrictive, but we just saw that the machines can not only compute the same functions, but in fact even the time and tape bounds are closely related. They just differ by a polynomial of small order. The concept of *polynomial equivalence* will be discussed later when we switch to emphasise the complexity aspects more and invent complexity classes.
In the following theorem we assume that input and output of the k-tape Turing machine is on tape 1 and that before and after the computation the other tapes are empty. This does obviously not have an effect on the number of computable functions, as for a machine without this property we could simply copy the contents of the other tapes after the content of tape 1. The tape- and time-complexity of this modified machine would just differ in a constant.

37 Theorem *If M is a $t(n) \geq n$ time bounded and $b(n)$ tape bounded k-tape Turing machine with input and result on tape* 1*, then there is an $O(b(n))$ tape bounded and $O(t(n) \cdot b(n))$ time bounded* 1*-tape Turing machine M^*, that simulates M.*

Note that the restriction that input and result must be on tape 1 does not restrict the set of computable functions. If parts of the results are also on other tapes, we can copy them to tape 1 afterwards – maybe separated by some separation symbols – and would get a result that is in a certain sense equivalent. Analogously input on tape 1 can be distributed over the other tapes. This would alter the time and tape complexities just by a constant.

Proof: For this proof we will also just give the ideas to understand the proof. The details are **important** but at this point not really interesting any more. Furthermore they can be filled in easily by each of you!

The idea to simulate different tapes on different tracks is surely not surprising. A problem is of course that while on different tapes the heads can move in different directions, there is only one head – and therefore only one direction – in case of a 1-tape Turing machine. So we must make sure that the head can read all the letters that the k heads read in one step. That is: they must be in the same cell.

Here we could also assume that M does not print a B to make some arguments a bit easier, but this is in fact not necessary, as we use tracks – so each cell ever visited during the main computation can be detected as it carries a character that stands for a subdivision into tracks – so even if there are blanks on the tracks, the cell can still be recognized as being visited.

We construct a machine with $k+1$ tracks. Important partial machines are $M_{L,i}, M_{R,i}$ for $1 \leq i \leq k$ that move track i one cell to the left ($M_{L,i}$) resp. right ($M_{R,i}$).

In order to have all letters that have to be read in the same cell, we use a *headcell* marked with an x on track $k+1$. Instead of moving e.g. the head on tape i one cell to the left, we move track i one step to the right, making sure that the same letter that head i reads in the Turing machine M stands on track i in the headcell of M^*.

					1	1	1	1	0	#	0	0	1					
								1	1	1	1	1	0	#	#	#		
		1	#	#	#	1	0	0	B’	B’								
								B’	B’	B’	1	0	0	0				
							B’	B’	0	0								
							1	1	1									
								x										

For all $d_1, \ldots, d_k \in \{L, N, R\}^k$ and states z of M we construct sub-machines $M_{z,d_1,\ldots,d_k}$ that move for each $1 \leq i \leq k$ track i one cell in the opposite direction of d_i and afterwards return to the headcell and change to state z.

It is possible, but more complicated to describe the $M_{z,d_1,\ldots,d_k}$ as stand-alone machines and combine them afterwards, as they have to change back to different states of the same machine and not always the same start state of the machine interpreting the letters in the way M does. In order to keep it simple, we will stay with the *submachines* – the difference are just details.

Then we can exactly like the k-tape Turing machine M read all necessary letters in one step and determine what to write and the next state. Then $M_{z,d_1,\ldots,d_k}$ simulates the movements of the heads (this needs time $O(b(n))$) and afterwards the machine is ready to simulate the next step of M.

At the end the tracks have to be erased and the head positioned on the first letter of the content.

This machine uses $O(b(n))$ tape ($2 \cdot b(n) + 2$ is an upper bound) and as each step of M can be simulated in $O(b(n))$ time, the time bound is $O(t(n) \cdot b(n))$. Subdividing into tracks and removing tracks can be done in linear time.

■

38 Corollary *Each k-tape Turing machine can be simulated by a 1-tape Turing machine that never uses cells left of the original positions of the head. The time and tape complexities just differ by a polynomial.*

So in fact a 1-tape Turing machine with a tape that is only infinite in one direction can do the same as a k-tape Turing machine with tapes that are unbounded in both directions – and that (in a certain sense) almost as efficiently.
In the literature one finds many more definitions of Turing machines that are studied – e.g.:

- Turing machines that have an extra tape for input and output.
- Turing machines that have several heads on each tape.
- Turing machines that have multidimensional *tapes* (compare the reasoning of Turing in the beginning of this chapter).
- Turing machines that have a tape with the structure of an infinite tree.

All these models (and many more) give the same class of *Turing-computable* functions.

34 Exercise *Make Turing's reasoning that a 1-dimensional structure of the tape can* fulfil the same function *as a two-dimensional one a bit more formal. Give a definition of when you would say that a Turing machine with a 1-dimensional tape and a generalised Turing machine with a 2-dimensional tape compute the same function and sketch a proof that both models can compute the same functions.*
A Turing machine in our sense can of course never have an input or a result that has the shape of a square. This is of course a problem that has to be taken care of in the interpretation of equivalence. . .

35 Exercise *An exercise that is a bit vague:*
We are always talking about arbitrary finite alphabets and sometimes use huge alphabets when we are working with Turing machines with tracks. But Turing said also something about alphabets. . .
Assume now that you don't have arbitrarily large alphabets, but a definition of a Turing machine that has as alphabet just $\{0,1\}$ *(but is otherwise the same). An* empty *tape is a tape with all* 0*. Can you still compute* the same *functions?*
Define exactly what the same functions *means and prove that you can still compute the same functions. With a reasonable definition of* the same *this should be possible. Determine also the difference in time and tape complexity.*

36 Exercise *Prove or disprove:*

- *If we allow infinite alphabets for Turing machines then for every Turing machine T with a (countably) inifinite alphabet there is a Turing machine T' with a finite alphabet so that if T starts on an input from $\{0,1\}^*_\_$ and has an output from $\{0,1\}^*_\_$ then T' started on the same input has the same output.*

- *If we allow a (countably) infinite number of states for Turing machines then every Turing machine T' with an inifinite number of states can be simulated by a Turing machine T' with a finite number of states.*

37 Exercise *Given a k-tape Turing machine M with alphabet A and an alphabet A' with $\{0,1\} \subseteq A' \subseteq A$. Prove that there is a k-tape Turing machine M' with alphabet A' so that for all inputs from A'^* for which the output of M is also in A'^* we have that the result of M and M' are the same.*

4 Computability and not computable functions

4.1 Turing computability

Notation
For $n \in \mathbb{N}$ the binary representation of n without leading zeros is $\text{bin}(n)$.
So bin() is a function $\mathbb{N} \to \{0,1\}^+$.
We define $\text{bin}^{-1} : \{0,1\}^+ \to \mathbb{N}$ as
$\text{bin}^{-1}(w_1, \ldots, w_s) := \sum_{i=1}^{s} w_i \cdot 2^{s-i}$.
Note that $\text{bin}^{-1}(\text{bin}(n)) = n$, but bin^{-1} is also defined for elements of $\{0,1\}^+$ that are not in the image of bin().

39 Definition *A Turing machine M with alphabet A defines a function $f_M : A_\_^* \to A_\_^*$:*
For $w \in A_\_^*$ let $f_M(w)$ be the content of tape 1 after the computation started with w on tape 1 and otherwise empty tapes. If M does not stop, or the content is not in $A_\_^*$, $f_M(w)$ is undefined.
For $r \in \mathbb{N}$, $r > 0$ let $f_M^r : \mathbb{N}^r \to \mathbb{N}$ be defined as follows:

$$f_M^r(x_1, \ldots, x_r) := \begin{cases} \mathit{bin}^{-1}(y) & \text{if } M \text{ started on} \\ & \mathit{bin}(x_1)\# \ldots \#\mathit{bin}(x_r) \\ & \text{on tape 1 and the other} \\ & \text{tapes empty, stops with} \\ & \text{content } y \in \{0,1\}^+ \text{ on tape 1} \\ \text{undefined} & \text{else} \end{cases}$$

40 Definition *For a given alphabet A a function $f : A_\_^* \to A_\_^*$ is called Turing computable, if there exists a Turing machine M with alphabet $A' \supseteq A$ so that $f = f_M|_{A_\_^*}$ (With as usual $f|_{A_\_^*}$ denoting the restriction to the set $A_\_^*$).*
A function $f : \mathbb{N}^r \to \mathbb{N}$ is called Turing computable, if there exists a Turing machine M with $f = f_M^r$.

It does not make a difference whether the definition requires a regular machine with $f = f_M$ resp. $f = f_M^r$ or whether a not necessarily regular machine is considered sufficient. Nevertheless it is not completely trivial to see this as the content after the computation need not be regular.

38 Exercise • *Show that you get an equivalent definition of* Turing computable *if you require that a regular Turing machine M exists instead of requiring an arbitrary machine. Note that the content of tapes number 2 or larger can contain blanks.*

- *Show that you get an equivalent definition of* Turing computable functions $f : \mathbb{N}^r \to \mathbb{N}$ *if you require that a Turing machine* M *with* $f = f^r_M$ *exists that – if started on a regular input – never stops with a content* $y \notin \{0,1\}^+$ *on tape* 1.

  For the following lemmas we assume that the Turing machines that compute the functions f, g, h *all have this property.*

41 Lemma *The functions* $c^r_s(), o()$ *and* $p^r_i()$ *are Turing computable for all* $r, s, i \in \mathbb{N}$, $r > 0$, $1 \leq i \leq r$.

Proof: We have constructed machines computing these functions already in earlier exercises.

■

42 Lemma *If* $f : \mathbb{N}^r \to \mathbb{N}$ *and* $g_1, \ldots, g_r : \mathbb{N}^m \to \mathbb{N}$ *are Turing computable, then also* $h : \mathbb{N}^m \to \mathbb{N}$ *defined as* $h(x) := f(g_1(x), \ldots, g_r(x))$ *for all* $x \in \mathbb{N}^m$.

Proof: Assume that $F, G_1, \ldots, G_r$ are regular k-tape Turing machines, computing $f, g_1, \ldots, g_r$ (and either don't stop or have a number as result). We may assume that for all machines the same k has been chosen. Due to Theorem 37 we may even assume that $k = 1$. But as the proof is as easy for arbitrary k, we'll just stick to that.

A regular machine copying the regular content of tape i to tape j and removes an earlier regular content of tape j if that wasn't empty, is easy to describe. We call such a machine $M_{i \to j}$.

A machine that attaches the regular content of tape i to the end of the regular content of tape j, separated by a # in case tape j was not empty, is as easy to describe. We call it $M_{i>>j}$.

We will now use these machines to define a $k+2$-tape Turing machine M_i computing g_i without modifying the content of tape 1 and writing the result on tape 2:

M_i is defined as:

$M_{1\to 3} \to G_i(3, \ldots, k+2) \to M_{3>>2} \to M_{\text{remove}}(3) \to \cdots \to M_{\text{remove}}(k+2)$

After M_1 is ready, the input $x \in \mathbb{N}^m$ is still on tape 1, $g_1(x)$ is on tape 2 and the tapes 3… $k+2$ are empty.

Now we define H as

$M_1 \to \cdots \to M_r \to F(2, \ldots, k+1) \to M_{2\to 1}$.

It is easy to see that $f_H = h$.

■

43 Lemma *If $f : \mathbb{N}^r \to \mathbb{N}$ and $g : \mathbb{N}^{r+2} \to \mathbb{N}$ are Turing computable, then also the function $h : \mathbb{N}^{r+1} \to \mathbb{N}$ with the properties*

1.) $h(0, x) = f(x) \quad \forall x \in \mathbb{N}^r$

2.) $h(n + 1, x) = g(n, h(n, x), x) \quad \forall n \in \mathbb{N}, x \in \mathbb{N}^r$

Proof: Assume F, G and k to be defined analogously to the previous proof. We will now describe a $k + 4$-tape Turing machine computing h:

The Turing machine $\bar{M}$ is defined as

$M_{4\to5} \to M_{3>>5} \to M_{2>>5} \to G(5, \ldots, k+4) \to M_{5\to3} \to M_{\text{remove}}(5) \to \cdots \to M_{\text{remove}}(k + 4) \to M_{-1}(1) \to M_{+1}(4)$

This machine computes g for the different recursion depths. For this machine to work correctly we have to guarantee that on tapes $2, 3, 4$ the corresponding parameters are to be found. In each depth we want to have the recursion parameter d on tape 4, $h(d, x)$ on tape 3 and x on tape 2.

The recursion is started with $d = 0$ and after each iteration d is incremented until we reach the n of the input.

Regular machines $M_{\text{Head } i}$ and $M_{\text{Tail } i}$ that work on tape i and remove everything except the first number (Head) or just the first number and maybe a following # if present (Tail) are easy to describe.

The machine H computing h is defined as

$M_{1\to2} \to M_{\text{Head } 1} \to M_{\text{Tail } 2} \to M_{2\to3} \to F(3, \ldots, k + 2) \to M_{c,0}(4) \to M_{\text{remove}}(5) \to \cdots \to M_{\text{remove}}(k + 2) \to \bar{M}_{\text{while tape } 1\neq0} \to M_{3\to1}$

Tape 1 counts iterations and tape 2 contains x during the computation. If the i-th loop is running, tape 4 contains $i - 1$ and tape 3 the value of $h(i - 1, x)$. In the beginning of the loop, tape 5 is initialised to $\text{bin}(i - 1)\#\text{bin}(h(i - 1, x))\#\text{bin}(x_1), \ldots, \text{bin}(x_r)$. Then G is computed for this set of parameters and the tapes are prepared for the following iteration.

■

44 Lemma *If f is Turing computable, then also μf.*

Proof: Assume k,F to be as in the previous proofs. We will compute μf by computing $f(n, x)$ starting with $n = 0$ and then incrementing n step by step until the result is 0. If $f(n, x) \neq 0$ for all $n \in \mathbb{N}$, the machine will never stop.

In case $f(n, x)$ is undefined for some n that is smaller than the first zero, the machine will also not stop, giving an undefined result for this x, which is correct.

The machine $\bar{M}$ computing $f(n, x)$ for the different values of n is defined by (n on tape 3, x on tape 2):

$M_{3\to 4} \to M_{2>>4} \to F(4, \ldots, k+3) \to M_{4\to 1} \to M_{\text{remove}}(4) \to \cdots \to M_{\text{remove}}(k+3) \to M_{+1}(3)$

In the last part the number on tape 3 is incremented by 1 for the next iteration. A Turing machine computing μf is

$M_{1\to 2} \to M_{c,1}(1) \to M_{c,0}(3) \to \bar{M}_{\text{while tape } 1\neq 0} \to M_{-1}(3) \to M_{3\to 1}$

■

If we combine all these lemmas, we have:

45 Theorem *Each μ-recursive function is Turing computable.*

What we will show now is that the reverse is also valid, so that in fact those two definitions of computability, that we have seen so far, lead to the same classes of computable functions. We will assume that except for the stp states all transitions are defined. If this is not the case, we can replace undefined transitions by transitions to the same state, printing the letter just read and not moving. This modified machine will also not stop.
We will now see something that one could describe as *simulating Turing machines with μ-recursive functions.*
First we will code configurations as numbers, but as μ-recursive functions only know numbers while configurations are strings, we first have to code strings as numbers:

46 Definition *Assume A to be an alphabet and Z a set of states. Note that we always require A and Z to be disjoint. $|A \cup Z| =: p$. Assume $A \cup Z$ to be ordered in an arbitrary way, so $A \cup Z = \{b_1, \ldots, b_p\}$. In order to make things easier in what follows we assume b_1 is the letter 0 and b_2 is the letter 1.*
Then we define $\psi : (A \cup Z)^+ \to \mathbb{N}$ as
$\psi(b_i) := i$ for all $b_i \in A \cup Z$

and $\psi(w_1 w_2 \ldots w_n) := \sum_{i=1}^{n}(\psi(w_i)(p+1)^{n-i})$ *for all* $w = w_1 w_2 \ldots w_n \in (A \cup Z)^+$

In fact this is a representation that we are used to. If we take a string with letters $0 = b_1, 1 = b_2 \ldots 9 = b_{10}$ and do the little change that we start the indexes with 0 instead of 1 and $p = 9$ is the largest index with this new numbering, then our value $\psi()$ would simply be the natural number that this decimal representation stands for.
The only problem with this choice of indexes is that the coding would not be injective – strings where the only difference are leading zeros, would be encoded by the same number. In order to avoid this and obtain an injective functions we have chosen to start with index 1 and obtain a coding with $\psi(b_i) > 0$ for all b_i.

11 Example $A \cup Z := \{0, 1, \#, B, z_0, z_e\}$, $\psi(0) = 1$, $\psi(1) = 2$, $\psi(\#) = 3$, $\psi(B) = 4$, $\psi(z_0) = 5$, $\psi(z_e) = 6$ *and* $\psi(Bz_0110B) = 4 \cdot 7^5 + 5 \cdot 7^4 + 2 \cdot 7^3 + 2 \cdot 7^2 + 1 \cdot 7^1 + 4 \cdot 7^0 = 67228 + 12005 + 686 + 98 + 7 + 4 = 80028$

This coding has the advantage that all operations on strings can be simulated in a relatively easy way by primitive recursive functions working on the codes:

47 Lemma *Assume A and Z to be given.*
Then there are primitive recursive functions l, concat, prefix, suffix, first, last, select, so that for all $v, w \in (A \cup Z)^+$ *and all* $i \in \{1, \ldots, |v|\}$ *we have:*

1.) $l(\psi(v)) = |v|$

2.) $concat(\psi(v), \psi(w)) = \psi(vw)$

3.) $prefix(\psi(v), i) = \psi(v_1, \ldots, v_i)$

4.) $suffix(\psi(v), i) = \psi(v_i, \ldots, v_{|v|})$

5.) $first(\psi(v)) = \psi(v_1)$

6.) $last(\psi(v)) = \psi(v_{|v|})$

7.) $select(\psi(v), i) = \psi(v_i)$

Proof: In order to understand the following proof more easily it can help again to keep the similarity to decimal representations of natural numbers in mind.

1.) $\mathrm{l}(n) = \min\{m | m \leq n \text{ and } (p+1)^m > n\}$
Note that the **bounded** μ-operator is sufficient for this.

2.) $\text{concat}(n, m) = n \cdot (p+1)^{\text{l}(m)} + m$

3.) $\text{prefix}(n, i) = \text{div}(n, (p+1)^{\text{l}(n)-i})$

4.) $\text{suffix}(n, i) = \text{rest}(n, (p+1)^{\text{l}(n)-i+1})$

5.) $\text{first}(n) = \text{prefix}(n, 1)$

6.) $\text{last}(n) = \text{suffix}(n, \text{l}(n))$

7.) $\text{select}(n, i) = \text{first}(\text{suffix}(n, i))$

In order to be able to write up the following results and proofs more easily, we use the shorthand notations $[n, m]$ for $\text{concat}(n, m)$ and even shorter $[n_1, \ldots, n_k]$ for $[[[n_1, n_2], n_3], \ldots n_k]$.

■

Notation
For a Turing machine M and a configuration C of M that is not an end configuration, we write $\Delta(c)$ for the successor configuration c' with $c \vdash c'$.

48 Lemma *For each* 1*-tape Turing machine* M *there is a primitive recursive function* $\tilde{\Delta} : \mathbb{N} \to \mathbb{N}$ *with* $\tilde{\Delta}(\psi(c)) = \psi(B\Delta(c)B)$ *for all configurations* c *that are not an end configuration and start and end with at least two blanks.*

The requirement that there must be two blanks in the beginning and the end is not really necessary, but makes some steps in the proof easier as you don't have to look at too many different cases. One has to look e.g., only at the first case in Definition 29 and one has always a non-empty prefix and suffix of the middle part. Lemma 47 is only stated for non-empty strings, but could also be generalized. The intention of the two explicit blanks in the beginning and end of $B\Delta(c)B$ is just to guarantee this prerequisite for the next steps.

Proof: First we will describe a function that determines the place where the state is situated in the configuration:

As the set Z and therefore also $\psi(Z)$ are finite, $\chi_{\psi(Z)}$ is primitive recursive (that was Exercise 17).

Define with $f(i, n) = 1 \dot{-} \chi_{\psi(Z)}(\text{select}(n, i))$ (note that the order of parameters of select() and $f()$ is different)

$z(n) = \mu_b f(n, n))$

Then we have

$$z(n) := \begin{cases} min\{i | i \leq n \text{ and } \mathrm{select}(n,i) \in \psi(Z)\} & \text{if the set is not empty} \\ 0 & \text{else} \end{cases}$$

As the bounded μ-operator was used, the function $z()$ is primitive recursive.

The function $z()$ computes the position of the state in the configuration. So if $a, b \in A^*$, $z \in Z$, then $z(\psi(azb)) = |a| + 1$. For this step we don't need the blanks in the beginning and the end.

The following functions *cut* the parts out of the configuration that will not be changed in the next step:

Define

- $\mathrm{firstpart}(n) := \mathrm{prefix}(n, z(n) \dot{-} 2)$ – the beginning of the configuration until one letter before the state
- $\mathrm{lastpart}(n) := \mathrm{suffix}(n, z(n) + 2)$ – the part two letters after the state until the end
- $\mathrm{middle}(n) :=$
  $[\mathrm{select}(n, z(n) \dot{-} 1), \mathrm{select}(n, z(n)), \mathrm{select}(n, z(n) + 1)]$
  – the middle part: the letter before the state, the state and the letter after the state

The next function simulates the definition of $\vdash$ in Definition 29 applied to the middle part. This part contains just 3 symbols, so there are just finitely many combinations possible. So the following function is primitive recursive as it differs from 0 for only finitely many inputs.

$$\tilde{\delta}(n) = \begin{cases} \psi(bcz') & \text{if there are } b, a \in A, z \in Z \text{ so that } n = \psi(bza) \text{ and } \delta(z,a) = (z', c, R) \\ \psi(bz'c) & \text{if there are } b, a \in A, z \in Z \text{ so that } n = \psi(bza) \text{ and } \delta(z,a) = (z', c, N) \\ \psi(z'bc) & \text{if there are } b, a \in A, z \in Z \text{ so that } n = \psi(bza) \text{ and } \delta(z,a) = (z', c, L) \\ 0 & \text{else} \end{cases}$$

The function $\tilde{\Delta}$ we want to construct, can now be given as
$\tilde{\Delta}(n) = [\psi(B), \mathrm{firstpart}(n), \tilde{\delta}(\mathrm{middle}(n)), \mathrm{lastpart}(n), \psi(B)]$.

■

49 Lemma *For each* 1*-tape Turing machine M there is a primitive recursive function end* : $\mathbb{N} \to \mathbb{N}$ *so that for each configuration c of M we have:*

$$\textit{end}(\psi(c)) = \begin{cases} 0 & \textit{if } c \textit{ is end configuration} \\ 1 & \textit{else} \end{cases}$$

It is not important what the value is for numbers not representing configurations.

Proof: With the notation of the previous proof, with E the set of stop states, the following function does the job:

$\text{end}(n) = 1 \dot{-} \chi_{\psi(E)}(\text{select}(n, z(n)))$

■

The strategy to follow is now clear: we will repeatedly apply the function $\tilde{\Delta}$ to simulate the working of the Turing machine. Using the function end() we can test whether an end configuration has been reached. The values of end() (that is: when 0 and when not 0) are chosen in a way that the μ-operator determines the number of iterations after which an end configuration is reached.

Or precisely:

50 Lemma *For each* 1*-tape Turing machine M there is a μ-recursive function endconf* : $\mathbb{N} \to \mathbb{N}$ *so that for each configuration c of M we have:*

$$\textit{endconf}(\psi(c)) = \begin{cases} \psi(B \dots B c' B \dots B) & \textit{if } M \textit{ started on configuration } c \\ & \textit{stops with configuration } c' \\ \textit{undefined} & \textit{else} \end{cases}$$

Proof: First we define the function $D : \mathbb{N}^2 \to \mathbb{N}$ as

$D(0, m) := m$

$D(n+1, m) := \tilde{\Delta}(D(n, m))$

For a configuration c and an $n \in \mathbb{N}$ that is not larger than the number of steps necessary to reach an end configuration starting from c, $D(n, \psi(c))$, we have that $D(n, \psi(c))$ is the code of a configuration $B \dots B c' B \dots B$ with the configuration c' the result of n steps of the machine M applied to c.

Now we define $f(t, n) = \text{end}(D(t, n))$ and $T(n) := \mu f(n)$. Then T is μ-recursive and

$$T(n) = \begin{cases} \min\{i | D(i,n) \text{ is the code} & \text{if such an } i \text{ exists} \\ \qquad \text{of an end configuration}\} & \\ \text{undefined} & \text{else} \end{cases}$$

So endconf$(\psi(c)) := D(T(\psi(c)), \psi(c))$ is defined if and only if the Turing machine started with the start configuration c stops. In that case $D(T(\psi(c)), \psi(c))$ is the code of the end configuration with possibly a number of extra blanks in the beginning and in the end.

■

All that is left to do is to transform the input to the code of a start configuration and decode the code of the end configuration to get the result – and all that with primitive recursive functions.

51 Lemma *For each* 1*-tape Turing machine M and for all $r \in \mathbb{N}$ there are primitive recursive functions input* : $\mathbb{N}^r \to \mathbb{N}$ *and output* : $\mathbb{N} \to \mathbb{N}$ *so that for all $m \in \mathbb{N}$ and $(x_1, \ldots, x_r) \in \mathbb{N}^r$ we have*
input$(x_1, \ldots, x_r) = \psi(BBz_0 bin(x_1)\#bin(x_2)\# \ldots \#bin(x_r)BB)$
and
output$(\psi(B \ldots Bz bin(m)B \ldots B)) = m$

Proof: We will use that if $g : \mathbb{N}^{r+1} \to \mathbb{N}$ is primitive recursive, then also $h : \mathbb{N}^{r+1} \to \mathbb{N}$ defined as

$h(n,x) := \sum_{i=0}^{n} g(i,x)$

This follows easily as $h(0,x) = g(0,x)$ and $h(n+1,x) = g(n+1,x) + h(n,x)$.

We define position : $\mathbb{N}^2 \to \mathbb{N}$ as position$(n,m) := \text{rest}(m,(2^n))/2^{n \dot{-} 1}$.

So the result of position(n,m) is the n-th position in the binary representation of m.

This means that input$'$: $\mathbb{N}^2 \to \mathbb{N}$ defined as

input$'(n,m) := \sum_{i=1}^{n} \psi(\text{position}(i,m)) \cdot (p+1)^{i \dot{-} 1}$

is primitive recursive and input$'(\text{l}'(m), m)$ is $\psi(\text{bin}(m))$, if we define l$'()$ just as l(), but with base 2 instead of base $p+1$.

Now input() can be obtained by using concat() to concatenate the different parts:

input$(x_1,\dots,x_r) =$
$[\psi(B),\psi(B),\psi(z_0),\mathrm{input}'(\mathrm{l}'(x_1),x_1),\psi(\#),\dots,\psi(\#),\mathrm{input}'(\mathrm{l}'(x_r),x_r),\psi(B),\psi(B)]$

At last the function output():

The function decode : $\mathbb{N}\to\mathbb{N}$ defined as
$decode(n) := \sum_{i=1}^{\mathrm{l}(n)}((\mathrm{rest}(n,(p+1)^i)/(p+1)^{i\dot{-}1})\dot{-}1)\cdot 2^{i\dot{-}1}$

has the property that decode$(\psi(\mathrm{bin}(m))) = m$.

If we are now able to extract the number $\psi(\mathrm{bin}(m))$ from $\psi(B\dots Bz_e\mathrm{bin}(m)B\dots B)$ in a primitive recursive manner, we can give the function output().

Define first$_B : \mathbb{N}\to\mathbb{N}$ as

$$\mathrm{first}_B(n) := \begin{cases} \min\{i|i\le n \text{ and } \mathrm{select}(n,i)=\psi(B)\} & \text{if the set is not empty}\\ 0 & \text{else}\end{cases}$$

If n codes a string containing a B, first$_B(n)$ returns the position of the first B in the string.

We define extract : $\mathbb{N}\to\mathbb{N}$ as

extract$(n) :=$ prefix(suffix$(n, z(n)+1)$, first$_B$(suffix$(n,z(n)+1))\dot{-}1)$.

So extract() gives the middle part of the string containing the code of $\psi(\mathrm{bin}(m))$.

Now we can define output() as

output$(n) :=$ decode(extract(n))

■

39 Exercise *Show that the function first$_B(n)$ is primitive recursive.*

Of course the aim of the lemmas we have just proven is mainly the following important theorem:

52 Theorem *Each Turing computable function $f:\mathbb{N}^r\to\mathbb{N}$ is μ-recursive.*

Proof: $f(x_1,\dots,x_r) = \mathrm{output}(\mathrm{endconf}(\mathrm{input}(x_1,\dots,x_r)))$

■

Note that in some cases the same Turing machine can compute functions $f:\mathbb{N}^r\to\mathbb{N}$ for several different r – e.g. $f(x_1,\dots,x_r) = x_1+\dots+x_r$ can be computed by the same Turing machine for all r. For primitive recursive functions the r is fixed.

53 Corollary *(Kleene's normal form)*
If $f : \mathbb{N}^r \to \mathbb{N}$ is μ-recursive, then there are primitive recursive functions $p : \mathbb{N}^{r+1} \to \mathbb{N}$ and $q : \mathbb{N}^{r+1} \to \mathbb{N}$ so that
$f(x) = q(\mu p(x), x)$

Proof: Each μ-recursive function can be computed by a Turing machine . If this computation is simulated like in the proofs of the previous lemmas, you get exactly the form given in the corollary. Of course some of the primitive recursive functions have to be combined to obtain $p()$ and $q()$.

What is most important is that you can represent a μ-recursive function in a way that uses the unbounded μ-operator at most once!

■

54 Corollary *A function $f : \mathbb{N}^r \to \mathbb{N}$ is primitive recursive if and only if there is a primitive recursive function $t : \mathbb{N} \to \mathbb{N}$ and a $t(n)$ time bounded Turing machine M that computes the function $f()$.*

40 Exercise *Proof Corollary 54.*
This is an exercise that is ideal to understand the connections between the previous proofs. Maybe the exercise needs a bit more time than other exercises, but it is definitely worth the time!

41 Exercise *The fact that the Ackermann function needs very much time to compute the values, if you implement the recursive definition on a computer does not mean that there isn't a better algorithm to compute the Ackermann function. In* Datastructuren en Algoritmen 2 *you have seen examples where a recursive definition was extremely slow, while another implementation (e.g. using dynamic programming) was very fast.*
Can you use Corollary 54 to say something about the possible existence of fast Turing machines for computing the Ackermann function?
Can you say something about the existence of such machines without Corollary 54?

42 Exercise *Show that a function $\tilde{\Delta}$ with $\tilde{\Delta}(\psi(k)) = \psi(\Delta(k))$ also exists if you do not require that the configuration starts and ends with two blanks.*

43 Exercise *In Definition 25 the part where it is required that for all $n < m$ the function $f(n, x)$ is defined, is a bit ugly. Assume now that we do not require that but instead require in Definition 26, part c, only for total functions that $\mu f \in \mathcal{R}$. Show that we would get the same class $\mathcal{R}$ that we have now.*

4.2 Church's thesis

We have seen that the definitions of μ-recursive functions and Turing computable functions are not only different – they even emphasise completely different aspects of our intuition for computability. One approach starts from *functions we want to consider computable and ways to combine the functions* that we expect to preserve computability and one approach starts from the *process of doing a computation.* Nevertheless both approaches give exactly the same class of computable functions.
There are also other approaches to define *computability* – e.g. based on a model that is more similar to a usual computer, but all these approaches give the same class of computable functions.
Already in 1936 A. Church formulated the following thesis that is known under the name Church's thesis or Church-Turing thesis:

Church's thesis:
The class of *intuitively computable functions* is the class of functions that can be computed by a 1-tape Turing machine.

Of course one can't prove this thesis – the term *"intuitively computable functions"* is simply not well defined – that's after all our problem we started with.
One can easily give definitions of computability that don't give the same class (e.g.: *A function is called computable if and only if it is constant*), so all possible definitions of computability are obviously not equivalent. But all definitions that really tried to capture our intuition of computability lead to the same class of computable functions as Turing computability.
This justifies that from now on, we sometimes just say *computable* when we mean *Turing computable.*

4.3 Not computable functions

Our first example will be the famous busy beaver.

55 Definition *A* busy beaver *candidate with n states is a 1-tape Turing machine with n states (traditionally not counting the end state) and alphabet $\{0, 1, \#, B\}$ that writes only 1's en B's and stops when started on an empty tape.*
A busy beaver *with n states is a* busy beaver *candidate with n states that writes at least as many 1s when started on an empty tape as each of the other* busy beaver *candidates with n states.*

The function $\Sigma(n)$ defined by Rado is the number of 1s written by a busy beaver with n states when started on an empty tape.

Of course in principle an alphabet containing just $\{1, B\}$ would be sufficient, but as for Turing machines we require $\{0, 1, \#, B\}$, we stick to that definition and won't introduce an alternative one here.
Sometimes the busy beaver is also defined as the machine doing the largest number of steps. Then the problem is obviously closely related to the Halting problem which we will (of course) also discuss. But defined this way, the problem looks (at least to me) less interesting and it is also not the *real* busy beaver function. For the function coming from this alternative definition, you will mostly find the notation $S(n)$.

44 Exercise *(Easy)*
Show that $\Sigma(n)$ is in fact well defined. Why does such a machine exist?

The only exact values known for $\Sigma(n)$ are $\Sigma(1) = 1$ and $\Sigma(2) = 4$ $\Sigma(3) = 6$ and $\Sigma(4) = 13$. That doesn't look impressive. For $n \geq 5$ only lower bounds are known: $\Sigma(5) \geq 4.098$ and $\Sigma(6) \geq 10^{18267}$ – so already here some serious growth starts...

56 Theorem *$\Sigma(n)$ is not computable.*

Proof:

A sketch of the proof:

We will only sketch the proof, as the function $\Sigma()$ itself won't be important for the rest of the lecture. We only give the proof as you have probably already heard of the busy beaver and might want to know the details and for the nice anology to the Ackermann function. We have already seen the basic technique of the proof and we will see it again before the end of the lecture – so getting used to it isn't bad anyway...

Analogously to the proof that the Ackermann function is not primitive recursive, we will show that $\Sigma(n) : \mathbb{N} \to \mathbb{N}$ grows faster than any total computable function $f : \mathbb{N} \to \mathbb{N}$.

Sometimes you need a bit experience to judge what a Turing machine can do and can not do, but we have already seen enough examples to know that the machines used here are possible. We also assume the result of Exercise 35.

Assume $f : \mathbb{N} \to \mathbb{N}$ to be a total computable function. Then also

$F(n) := \sum_{i=0}^{n}(f(i) + i^2)$

is computable (and total).

For F we have $F(n) \geq n^2$ and $F()$ is strictly monotonically increasing.

If you want to code a number in a binary like way with B for 0 and 1, you can't detect the end of the number, so we code the binary digits with more than one letter – e.g. code 0 as $1B$ and code 1 as 11. Let code(n) be a way to code a number on the tape using just B and 1 that still allows to detect the end of the number on a tape with otherwise just B.

Let M be a Turing machine that started on code(n) writes exactly n ones on the tape and afterwards stops.

One possible way for M to work when the coding is chosen as described, is to go to the right end of the input and add a 1 to the end of a string of ones separated from the number by BB. Then M must go back to the left and subtract one from the coded number until the result is zero. Such a machine only has to read and write 1 and B. Let n_M be the number of states of such a machine.

The machine M_c writes the code of the number $c \geq 2$ onto the tape. If k cells are used for a binary digit, this can surely be done by a machine with $k \cdot log_2(c) \leq k \cdot c$ states.

The machine M_F computes the function $F()$ on the codes. Let n_F be the number of states of M_F.

Now we analyse the following machine:

$M_c \rightarrow M_F \rightarrow M_F \rightarrow M$

This machine can operate with $k \cdot c + 2 \cdot n_F + n_M$ states and started on an empty tape it writes $F(F(c))$ ones onto the tape and stops.

So the definition of Σ gives:

$\Sigma(k \cdot c + 2 \cdot n_F + n_M) \geq F(F(c))$

Now choose c so large that $c^2 > k \cdot c + 2 \cdot n_F + n_M$. Then $F(c) \geq c^2 > k \cdot c + 2 \cdot n_F + n_M$ and therefore

$\Sigma(k \cdot c + 2 \cdot n_F + n_M) \geq F(F(c)) > F(k \cdot c + 2 \cdot n_F + n_M) > f(k \cdot c + 2 \cdot n_F + n_M)$

But as this reasoning is valid for any total computable function, $\Sigma()$ can not be computable – otherwise choosing $f = \Sigma$ would lead to a contradiction.

■

Note: $\Sigma(n)$ is a **total** not computable function.

45 Exercise *Discuss the following text that could be found on wikipedia. In that text* a busy beaver champion *is what we called* a busy beaver *and a* candidate *is most likely meant in the more general form as "a given Turing machine" and not as what we called a " busy beaver candidate".*

Radó went on to prove that there is no computable function that bounds $\Sigma()$*; that is, for any given computable function* $f()$*, there must be some* n *(and thus, one can show, infinitely many* n*), for which* $f(n) < \Sigma(n)$*. (A proof is given below.) In particular,* $\Sigma()$ *is itself non-computable.*
Moreover, this implies that it is undecidable whether a given candidate is a busy beaver champion (for if we could algorithmically determine whether or not a given candidate was champion, we could then determine the appropriate value of $\Sigma()$*).*

---

46 Exercise *(An easy one – just to show that something is well understood.) Show that for all* $c \in \mathbb{N}$ *there is a Turing machine computing a function* $f()$ *with the property that* $f(n) = \Sigma(n)$ *for all* $1 \leq n \leq c$ *and* $f(n) \neq \Sigma(n)$ *for all* $n > c$*.*

4.4 Decidability

The following functions don't have the problem that the values grow too fast – in fact the possible function values are just 0 and 1. We will now discuss decision problems. The result of a decision problem is either 1 ("yes") or 0 ("no"). An equivalent way of saying this is that we have a set $L \subseteq A^*$ (of strings for which the answer is yes) and want to decide whether for a given string w we have $w \in L$.
The basis of this theory lies in a problem that David Hilbert posed in the year 1900. It is known as Hilbert's 10th problem.

Question:
Is there an algorithm that given a diophantic equation as input can decide whether the equation has a solution in $\mathbb{N}$ or not?
A diophantic equation is a polynomial equation in several variables with integer coefficients where solutions must be integer numbers.
Important examples are equations of the form
$a_1 \cdot x_1^{b_1} + \cdots + a_n \cdot x_n^{b_n} = c_1 y_1^{d_1} + \cdots + c_m y_m^{d_m} + g$
with $a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_m, d_1, \ldots, d_m, g \in \mathbb{N}$

The question would then be whether the equation has the following property:
$\exists x_1, \ldots, x_n, y_1, \ldots y_m \in \mathbb{Z} \qquad a_1 \cdot x_1^{b_1} + \cdots + a_n \cdot x_n^{b_n} = c_1 y_1^{d_1} + \cdots + c_m y_m^{d_m} + g?$

But a diophantic equation can also have combinations of powers of the variables, like e.g.,

$7 \cdot x^2 \cdot y^5 + 9 \cdot y^1 \cdot z^7 - 3 \cdot x \cdot y \cdot z - 12 = 0$

This question was one of the motivations that lead to the theory of computability. The question was answered in 1970 by Juri Matijasevic: such an algorithm does not exist – or with other words: this problem is not decidable.

57 Definition *A set $L \subseteq A_{\_}^*$ (or $L \subseteq \mathbb{N}^r$ for some $r \in \mathbb{N}$, $r \geq 1$) is called* <u>*decidable*</u> *if the characteristic function χ_L is a* **total** *computable function.*

The fact that the number of subsets of $A_{\_}^*$ and $\mathbb{N}^r$ are uncountable while the number of Turing machines is countable implies immediately that there are undecidable sets.

12 Example *The set of all even numbers, all prime numbers and all finite sets are decidable.*
The set $\{(n,m) \in \mathbb{N}^2 | m = \Sigma(n)\}$ is not decidable. Why not? Compare this claim to Theorem 24.

58 Remark *A set L is decidable if and only if the complement $\bar{L}$ is decidable.*

Now we will discuss an especially important problem: the <u>halting problem</u>. You have probably already heard that it is not decidable, but now we will discuss the details.
In order to be able to formulate the problem in a mathematically rigorous way, we still need to introduce a few concepts:
The question is whether a Turing machine can decide whether another Turing machine started on some input will stop or not. As Turing machines only work with strings, we have to code Turing machines as strings in order to formulate the problem. That such a coding is possible is of course everything else than astonishing, as the way we write up Turing machines is also just a series of letters.

4.5 Gödelisation

We code a Turing machine only with letters from $\{0,1\}$. One could also choose another coding than the one given here and everything that follows could be done in an analogous way – we just have to fix one possible coding.
The first step is to code the machine in $\{0,1,\#\}$.
Let $M = (Z, A, \delta, z_0, E)$ be a 1-tape Turing machine.
We give numbers to the states starting with z_0 and to the letters starting with $0, 1, \#, B$ in this order. Our numbers start with 0, so $Z = \{z_0, \ldots, z_s\}$, $A = \{a_0 = 0, a_1 = 1, a_2 = \#, a_3 = B, \ldots, a_t\}$.
The directions L, N, R are numbered in the order $r_0 := N$, $r_1 := R$, $r_2 := L$.

Now each rule $\delta(z_i, a_j) = (z_k, a_l, r_m)$ with $z_i \notin E$
is assigned the word
$w_{ijklm} := \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(k)\#\text{bin}(l)\#\text{bin}(m)$.
Writing these words after each other, one gets a *complete* code of the Turing machine – up to some notion of *equivalence*: Z is the set of all states that occur in one of the words, A is the set of all letters that occur and E is the set of all states for which no word describes a transition function. Later on we will make this more explicit.

13 **Example** *The (trivial) Turing machine* $M^c = (Z, A, \delta, z_0, E)$ *with* $Z = \{z_0, z_1\}$,$A = \{0, 1, \#, B\}$,$E = \{z_1\}$ *and*
$\delta(z_0, 0) = (z_0, 1, R)$
$\delta(z_0, 1) = (z_0, 0, R)$
$\delta(z_0, \#) = (z_0, \#, R)$
$\delta(z_0, B) = (z_1, \#, N)$
gets coded as
##0#0#0#1#1##0#1#0#0#1##0#10#0#10#1##0#11#1#10#0
The way how this machine has to be written up, is uniquely fixed by the definition – except for the order in which the rules must be listed. For larger machines also the order of the states and the order in which letters from the alphabet are given numbers can vary.
Attention: *The code of a machine can be constructed in different ways – but from each of these codes a machine can be reconstructed that – in a certain sense – is the original machine. Compare Lemma 60.*

Now we will translate this code to a word in $\{0, 1\}^*$.
We define $h : \{0, 1, \#\}^* \to \{0, 1\}^*$ as $h(0) = 00$,$h(1) = 01$,$h(\#) = 11$ and for $w_1, \ldots, w_s \in \{0, 1, \#\}$
$h(w_1 \ldots w_s) = h(w_1) \ldots h(w_s)$

14 **Example** *The code of machine* M^c *is then*
111100110011001101110111110011011100110011011111001101001100110100
110111110011010111011101001100

59 **Definition** *For a 1-tape Turing machine* M *the code* $u \in \{0, 1\}^*$ *of* M *is called the* <u>*Gödelisation*</u> *of* M*. The natural number* n *with bin*$(n) = u$ *is called the* <u>*Gödel number*</u> *of* M*.*

You can compare the Gödel number with the program code of a normal computer.

60 Lemma *If two machines M and M' have the same Gödelisation and $v \in \{0,1,\#\}^*$, then M stops started on v if and only if M' does. If $f_M(v) \in \{0,1,\#\}^*$ then $f_M(v) = f_{M'}(v)$.*

47 Exercise *The principle of this proof is surely clear, but it is important that you can really write it up! So write a detailed proof of this Lemma up as some homework. Do this in so much detail like you would expect from a thesis or a book giving the proof.*

As the other letters are coded in an arbitrary way, Lemma 60 would not be correct for results not in $\{0,1,\#\}^*$.

48 Exercise *Describe two machines M and M' with the same alphabet and two Gödelisations u of M and u' of M' so that $u = u'$ and there exists a $w \in A$ so that M started on w stops and M' doesn't.*

The following lemma explains how you can decide whether a string $u \in \{0,1\}^*$ is or is not a Gödelisation of a 1-tape Turing machine. It would be easy to describe a Turing machine implementing an algorithm for this decision problem.

61 Lemma *A string $u \in \{0,1\}^*$ is a Gödelisation of a 1-tape Turing machine if and only if $u = h(u')$ for a string $u' \in \{0,1,\#\}^*$ with*

a.) *u' consists only of words of the form w_{ijklm}*

b.) *if w_{ijklm} and $w_{i'j'k'l'm'}$ are part of u', then $(i,j) \neq (i',j')$.*

It is of course necessary to prove this important lemma – but the proof is very easy. The direction where one has to show that a Gödelisation has these properties follows directly from the definition. For the other direction you have – starting from u or u' – to construct a Turing machine for which u is a possible Gödelisation:
Let u' be as in Lemma 61.
Define $M_u = (Z, A, \delta, z_0, E)$, $Z = \{z_0, \dots, z_t\}$, $A = \{a_0 = 0, a_1 = 1, a_2 = \#, a_3 = B, a_4, \dots, a_r\}$
with $t = \max\{i | \exists j,k,l,m,$ so that $w_{i,j,k,l,m}$ or $w_{k,j,i,l,m}$ are part of $u'\}$
$r = \max(\{j | \exists i,k,l,m,$ so that $w_{i,j,k,l,m}$ or $w_{i,l,k,j,m}$ are part of $u'\} \cup \{3\})$
$\delta(z_i, a_j) := (z_k, a_l, r_m)$ if $w_{i,j,k,l,m}$ is part of u'
The start state z_0 is the state z_0 in the list.
$E := \{z_i | i \leq t,$ and there are no $j,k,l,m,$ so that w_{ijklm} are part of $u'\}$

But we want more: we want for **each** string u in $\{0,1\}^*$ a Turing machine M_u. To this end we fix that if u is not a Gödelisation of a Turing machine, then M_u is some fixed machine that never stops – no matter what the input is.

62 Remark *If M is a 1-tape Turing machine, u a Gödelisation of M and $v \in \{0,1,\#\}^*$, then – started on v – M stops if and only if M_u stops. If in addition $f_M(v) \in \{0,1,\#\}^*$, then $f_M(v) = f_{M_u}(v)$.*

This follows immediately with Lemma 60 as u is a possible Gödelisation of M_u.
This corollary describes best and almost completely what the properties of a Gödelisation $M \to u(M)$ and of the "inverse" $u \to M_u$ are. If we have for example a number u that is the Gödelisation of a Turing machine M and we form the Gödelisation u' of M_u, then in most cases we will have $u' \neq u$ – depending on which Gödelisation is chosen (which order of the letters and states, which order of the w_{ijklm}). If you then form $M_{u'}$, in most cases you will have $M_{u'} \neq M_u$.

49 Exercise *Do the two machines $M_{u'}$ and M_u we just described have at least the same number of states?*

50 Exercise *Prove: For each n the set*

$\{u | M_u$ *is a busy beaver that writes* $\Sigma(n)$ *ones*$\}$

is decidable.

If the alphabet of a Turing machine M contains more letters than just $0, 1, \#, B$ – e.g. a letter X – then M and M_u don't even have the same alphabet and therefore can't compute the *same* functions. We could define when we consider functions to be equivalent based on bijections between the alphabets working on $\{0, 1, \#, B\}$ as the identity, but we can as well restrict our attention to functions $\{0,1,\#\}^* \to \{0,1,\#\}^*$.

4.6 The halting problem

63 Definition *The set $H_S := \{u \in \{0,1\}^* | M_u$ started on u stops$\}$
is called the special halting problem.*

64 Theorem *The special halting problem is not decidable.*

Proof: The proof uses the – already often used – diagonal argument.

Assume that χ_{H_S} is a total computable function. Then there is a 1-tape Turing machine H that computes χ_{H_S}. This implies that H stops for all inputs $u \in \{0,1\}^*$ and that for this u we have

$$f_H(u) := \begin{cases} 1 & \text{if } M_u \text{ started on } u \text{ stops} \\ 0 & \text{else} \end{cases}$$

Now we modify this machine and make a new machine $\bar{H}$ that works just like H in the beginning, but whenever H stops with a 0 on the tape, it replaces the 0 with a 1 and stops. If H stops with a 1 on the tape the new machine $\bar{H}$ enters an infinite loop – so it does not stop.

Assume that $\bar{u}$ is a Gödelisation of $\bar{H}$. What happens if $\bar{H}$ is started on $\bar{u}$?

$\bar{H}$ started on $\bar{u}$ stops $\Leftrightarrow f_H(\bar{u}) = 0 \Leftrightarrow M_{\bar{u}}$ started on $\bar{u}$ does not stop $\Leftrightarrow \bar{H}$ started on $\bar{u}$ does not stop.

But this is a contradiction – so H_S is not decidable.

■

The decidability of the following problems is investigated by using a technique called reduction. This is a technique that we will see quite often in the remainder – also when we will not discuss whether a function is computable, but how efficiently this is possible. You have possibly already seen this technique in datastructures and algorithm courses as it is also used for practical purposes to *translate* a new problem to an old, already known problem for which algorithms already exist.

65 Definition *If $L, L' \subseteq A_\_^*$ and $f : A_\_^* \to A_\_^*$ is a total computable function with the property that for all $w \in A_\_^*$ we have*
$w \in L \Leftrightarrow f(w) \in L'$
then L is called reducible to L' (by means of f). We write $L \leq_r L'$..

The symbol $\leq_r$ also shows something about the relation of the difficulties of the two problems: If you can solve L', you can also solve L. The other direction **can** also be true, but need not be true. So L is at most as difficult as L'.

66 Remark

a.) *If $L, L', L'' \subseteq A_\_^*$ and L is reducible to L' by means of f and L' is reducible to L'' by means of g, then L is reducible to L'' by means of $g \circ f$. So the relation $\leq_r$ is transitive.*

b.) *If $L, L' \subseteq A_{\_}^{*}$ and L is reducible to L' (so $L \leq_r L'$) and L is undecidable, then so is L'.*

Proof: a.) This follows directly from the definition.

b.) If $\chi_{L'}$ would be a total computable function, then also $\chi_{L'} \circ f$. But $\chi_{L'} \circ f = \chi_L$, so L would be decidable – a contradiction.

■

67 Definition

- *the set $H := \{u\#v | u, v \in \{0,1\}^*$ so that M_u started on v stops$\}$ is called the general halting problem.*
- *the set $H_0 := \{u | u \in \{0,1\}^*$ so that M_u stops when started on an empty tape$\}$ is called the halting problem on empty tape.*

68 Corollary *The general halting problem is not decidable.*

Proof: The special halting problem is obviously a special case of the general halting problem. The function f to reduce the special halting problem to the general one is just $f : \{0,1,\#\}^* \to \{0,1,\#\}^* : u \to u\#u$.

■

69 Corollary *The halting problem on empty tape is not decidable.*

Proof: We reduce the general halting problem to the halting problem on empty tape:

For each word $u\#v$ with $u, v \in \{0,1\}^*$ there exists a Turing machine $M(u,v)$ that – started on empty tape – first writes v onto the tape and then starts from the first letter in v working just like M_u.

Though the details are surely everything else but easy, it is clear that there is a Turing machine that can compute the Gödelisation $\bar{g}(u,v)$ of the machine $M(u,v)$ starting with $u\#v$.

In fact one can in principle even describe that machine! One has to first determine a Gödelisation of a Turing machine that writes v (starting for example with the last letter so that you are at the beginning of v when all letters are written) and then changes to the start state of M_u. Then all that is left to do is to change the numbers of the states in M_u and compute the Gödelisation of the whole new machine.

The function $g : \{0,1,\#\}^* \to \{0,1,\#\}^*$ is defined as:

$$g(w) := \begin{cases} \bar{g}(u,v) & \text{if } w \text{ is of the form } u\#v \text{ with } u,v \in \{0,1\}^* \\ 0 & \text{else} \end{cases}$$

As $M(u,v)$ started on an empty tape stops if and only if M_u started on v stops, we have $H \leq_r H_0$.

■

51 Exercise *For practical purposes you are of course mainly interested in* programs *of a bounded size. Translating this to Gödelisations, it means that we are mainly interested in Gödelisations that are bounded from above.*
Given a natural number b. Is the subset of the halting problem on empty tape that only contains Gödel numbers of at most b decidable? First write the set in a formal way, that is: $H_{0,b} := \{\dots\}$.

In fact all these results are a special case of a surprisingly general theorem of H.G. Rice. Informally one can say that there is almost no property of a function so that you can algorithmically determine whether an arbitrary given Turing machine computes a function that has this property. This means that it is e.g. not possible to determine whether the computed function is constantly 0 for all inputs. It is also not possible to determine whether there is at least one input for which the machine stops, etc.
In this context *not possible to determine algorithmically* means that there is no Turing machine that can decide that for **all** Turing machines. For a single fixed Turing machine that is of course always possible and for some given set of Turing machines that is sometimes possible.

52 Exercise *What does* "For a single fixed Turing machine that is of course always possible!" *mean exactly?*
Formulate exactly what that means (with the property "computes the constant function 0"*) – that is: write exactly which set is* always decidable *– and prove the result.*

But of course we will first give an exact formulation of the theorem of Rice and won't try to work with a vague one:

70 Definition *Let R be the set of all Turing computable functions* $\{0,1,\#\}^* \to \{0,1,\#\}^*$.

71 Theorem *(Rice)*
Let $S \subseteq R$. *The set* $C(S) := \{u | f_{M_u} \in S\}$ *is decidable if and only if* $S = \emptyset$ *or* $S = R$.

Proof: For an arbitrary given S with $S \neq \emptyset$ and $R \setminus S \neq \emptyset$ we either reduce the halting problem H_0 or its complement H_0^c to $C(S)$.

Let Ω be the function that is undefined for all inputs.

Let us first assume that $\Omega \in S$.

We reduce H_0^c to $C(S)$:

As $S \neq R$ there is also some function f_{S^c} in $R \setminus S$. Let M_{S^c} be a Turing machine computing f_{S^c}.

For each $w \in \{0,1\}^*$ there is a Turing machine $M(w)$ that started on each input $y \in \{0,1,\#\}^*$ first ignores the input and just works as M_w started on an empty tape (e.g. on another track).

If M_w stops, $M(w)$ will have completed this part of the job at some time. After having completed it, $M(w)$ continues in the same way as M_{S^c} started on y.

This means: Depending on whether M_w started on empty tape stops or not, the function computed by $M(w)$ is either Ω or f_{S^c}. $M(w)$ always computes the same function – either always Ω or always f_{S^c} – **not** once Ω once f_{S^c} depending on the input y.

We will again not give the details of a machine that for a given w computes the Gödelisation $u(M(w))$ of $M(w)$, but from our experience with Turing machines we *know* that it is possible to describe such a machine and that therefore the function $g : w \to u(M(w))$ is a total computable function.

So for an arbitrary input $w \in \{0,1\}^*$ we have that $f_{M_{g(w)}} \in S$ (or to be precise: $f_{M_{g(w)}} = \Omega$) and therefore $g(w) \in C(S)$ **if and only if** $w \in H_0^c$, so $H_0^c \leq_r C(S)$.

The case $\Omega \notin S$ can be proven completely analogously or by using that $C(S)$ is decidable if and only if $C(S^C)$ is decidable.

■

53 Exercise *Prove the following theorem or prove that it is wrong:*
Theorem(?):
Let $S \subseteq R$. Then the set

$$C(S) := \{u| \quad f_{M_u} \in S \text{ and among all } M \text{ computing } f_{M_u} \\ M_u \text{ has the smallest time complexity.}\}$$

is decidable if and only if $S = \emptyset$ or $S = R$.

Here M_u has a smaller time complexity than $M_{u'}$ means that there is an n so that for all inputs with length $n' \geq n$, M_u does less steps than $M_{u'}$. It is not clear whether for a given $f() \in S$ a Gödelisation u so that M_u has the smallest time complexity does exist. If it does not exist for a certain function, the set is empty for that function.

54 Exercise *Let $f \in R$. For which functions f is the set*

$$C(f) := \{u| \quad f_{M_u} = f \text{ and there does not exist a shorter Gödelisation than } u \text{ of a machine } M \text{ computing } f\}$$

decidable?

55 Exercise *Let $f \in R$. For which functions f is the set*

$$C(f) := \{u| f_{M_u}(x) = f(x) \text{ for only finitely many values of } x\}$$

decidable?
Note that $f_{M_u}(x) = f(x)$ means that both functions are defined for the input x and that their value is the same.

The following definition gives a weaker concept than *being decidable*. It can be compared with the concept of being countable – only that one requires the enumerating function to be computable.

4.7 Recursive enumerability

72 Definition *Given an alphabet A. A set $L \subseteq A^*_-$, resp. $L \subseteq \mathbb{N}^r$ for some $r \in \mathbb{N}, r > 0$ is called $\underline{\text{recursively enumerable}}$ if $L = \emptyset$ or there exists a total computable function $f : A^*_- \to A^*_-$ (resp. $f : \mathbb{N}^r \to \mathbb{N}^r$) with $f(A^*_-) = L$ (resp. $f(\mathbb{N}^r) = L$). In this case we say that f $\underline{\text{enumerates}}$ the set L.*

73 Theorem *A set L is decidable if and only if L* **and** *L^c are recursively enumerable.*

Proof: Assume first that $L \subseteq A^*_-$ is decidable. In case $L = \emptyset$ or $L = A^*_-$ the result follows directly from the definition. So assume $L \neq \emptyset$ and $L \neq A^*_-$, $v \in L$,$\bar{v} \notin L$.

Define

$$f_L(w) := \begin{cases} w & \text{if } \chi_L(w) = 1 \\ v & \text{else} \end{cases}$$

$$f_{L^c}(w) := \begin{cases} w & \text{if } \chi_L(w) = 0 \\ \bar{v} & \text{else} \end{cases}$$

As χ_L is a total computable function, f_L enumerates the set L and f_{L^c} enumerates the set L^c.

Assume now that f, f_c are two total computable functions enumerating L resp. L^c and that M_f, resp. M_{f_c} are the corresponding Turing machines – only that they are slightly modified as they expect the input on track 2, then copy track 2 to track 3 and do the computations only on track 3.

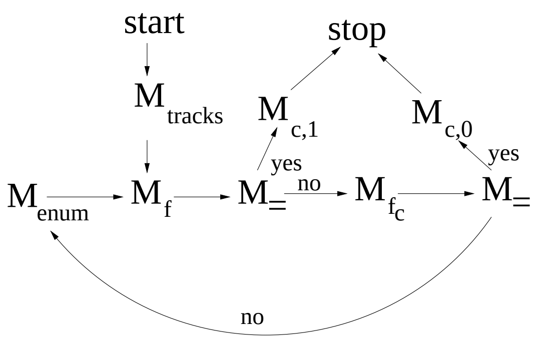
M_{tracks} subdivides the tape in 3 tracks writing the input on track 1 and keeping the other 2 tracks empty in the beginning.

Now one can *easily* make a machine M_{enum} that for a given order of letters in A started on a word in $A_\_^*$ produces the lexicographically next word of the same length or – in case we already have the lexicographically largest word of that length – the lexicographically first word that is one letter longer. The machine does this on track 2 without modifying the contents of track 1. Afterwards the result is copied on track 3.

The machine $M_=$ tests whether the contents of track 1 and 3 are equal and goes to the final state *yes* if this is the case and *no* otherwise.

For $i \in \{0, 1\}$ the machine $M_{c,i}$ removes the content of the tape and writes the constant i on the tape.

The machine M works as follows:



Each $w \in A_\_^*$ is the image of f or f_c applied to some w_o. As soon as w_o is generated by M_{enum} (and that is the case after a finite number of steps), the machine will end in a final state in the next step. So M

computes a total function and the result is 1 if the input is in L and 0 otherwise. So M computes $f_M = \chi_L$.

■

While the *universal primitive recursive function* we discussed in Exercise 25 does not exist, such a Turing machine does exist. It will be a useful tool in the following proofs.

74 Theorem and definition

There exists a 1*-tape Turing machine* U *– called a* universal Turing machine *– so that for all* $u \in \{0,1\}^*$ *and* $v \in \{0,1,\#\}^*$ *we have:*

- *U started on $u\#v$ stops if and only if M_u started on v stops. If $f_{M_u}(v) \in \{0,1,\#\}^*$ then $f_U(u\#v) = f_{M_u}(v)$.*

If the number of cells visited by M_u started on v is s, t is the number of steps done during the computation and $l = |f_{M_u}(v)|$ is the length of the result. Then U can be built in a way so that it

- *visits $O(|u| \cdot s)$ cells*
- *makes $O(|u|^2 \cdot (|v|^2 + t + l^2))$ steps*

when started on $u\#v$.

Proof: To save some time we will not give the detailed proof, but (again) just a sketch. The proof does not contain unexpected difficulties or unexpected ideas, so that – knowing the sketch – the details can be easily worked out by everybody.

First we test whether u is a Gödelisation of a Turing machine by testing the properties in Lemma 61. If that is not the case, U enters an infinite loop.

Otherwise we subdivide the tape into three tracks. The first track always contains the contents of the tape – but the letters of the alphabet of M_u are coded in $\{0,1\}^*$ – with all the same length – and are separated from each other by the letter #. The second track contains the state, which also marks the position of the head. Sometimes it also contains the code of the letter that has to be read separated by a #. The third track contains the inverse image of u in $\{0,1,\#\}^*$ under the mapping h.

Now we simulate M_u in the following way:

- the following rule is searched (that is some part of the contents of track 3 beginning with ##bin(j)#bin(k) if the state coded on track 2 is z_j and the letter read is a_k. This search is performed by moving the contents on track 3. If no rule beginning with ##bin(j) is found, the simulation is done – we reached a final state. If such a rule is found, but none starting with ##bin(j)#bin(k) then the behaviour is undefined (just like in the machine coded by the Gödelisation) and our machine goes into an infinite loop.
- if ##bin(j)#bin(k) is found on track 3, the modifications given in this rule are done. This means that track 1 and 2 are modified and moved to mark the new position of the head. In some cases also track 3 has to be moved in order to be below the position of marker of the head and not having to search for it on the tape.
- in the end the tracks have to be cleared up and the codes of the letters in $\{0, 1, \#\}$ must be translated back.

With the experience we have with Turing machines it should be possible to fill in the details of this proof. Moving u during the simulations takes time $O(|u|^2)$ for each simulation step – this is the dominating and most important part. For the complexity bounds of the translations in the beginning and the end we use that $|u|$ is an (in most cases much too large) upper bound for the code of a letter, so that the input is translated to a length of $|v| * |u|$ and that the output has length $l * |u|$. Moving parts of the input and output during the translation is responsible for the squares in the bound. One could go for sharper bounds for the translations, but as the dominating part is the simulation anyway, this is not important.

■

Now we can prove the following – at first sight – surprising result:

75 Theorem *The halting problem $H := \{u\#v | u, v \in \{0, 1\}^*$ with M_u started on v stops$\}$ can be recursively enumerated.*

Proof: We describe a 2-tape Turing machine M with alphabet $A = \{0, 1, \# B\}$ that computes a total function $f : A_-^* \to A_-^*$ with $f(A_-^*) = H$.

So assume an input from A_-^* is given. First it is tested whether the input has the form $u\#v\#\text{bin}(i)$ with $u, v \in \{0, 1\}^*$ and $i \in \mathbb{N}$.

If that is not the case, the tape is erased and $u_0\#0$ is written as output on the tape with u_0 the Gödelisation of an arbitrary machine that stops when started on the input 0 (so $u_0\#0 \in H$).

If the input has the form $u\#v\#\text{bin}(i)$, the machine M copies the part $\text{bin}(i)$ onto tape 2 and removes it from tape 1. Then M_u is simulated on tape 1 and after each step of the simulation the machines subtracts 1 from the number represented on tape 2.

If the simulation on tape 1 finishes before the number on tape 2 is 0 (that is: M_u started on v stops after at most i steps), $u\#v$ is written on the (cleaned) tape and the machine stops. To this end $u\#v$ must be stored on one tape or track.

If the simulation on tape 1 is **not** ready before the number on tape 2 is 0, the machine writes again $u_0\#0$ on the cleared tape 1 as soon as the value 0 on tape 2 is reached.

It is obvious that all results $u'\#v'$ of the computations are in H. In fact it is not only so that each word $u\#v \in H$ has an inverse image in A_-^* (what would be enough to show that $f_M(A_-^*) = H$) – each word $u\#v \in H$ has even infinitely many inverse images in A_-^*: for each word $u\#v \in H$ there is a smallest j so that $f_M(u\#v\#\text{bin}(j)) = u\#v$ – and then $f_M(u\#v\#\text{bin}(i)) = u\#v$ for all $i \geq j$.

■

76 Corollary *The set H^c can not be recursively enumerated.*

77 Theorem *There is no set $T \subseteq \{0,1\}^*$ that can be recursively enumerated and $\{f_{M_u} | u \in T\}$ is equal to the set S_t of total computable functions.*

*Note that as a Gödelisation is not unique and as there are infinitely many machines computing the same function, there are also infinitely many sets T with $\{f_{M_u} | u \in T\} = S_t$. So this theorem is **much** stronger than just claiming that $T = \{u \in \{0,1\}^* | f_{M_u} \in S_t\}$ can not be recursively enumerated!*

Proof: We use the often seen diagonal argument

Assume that such a T exists and is enumerated by h and assume w.l.o.g. that the alphabet is $A = \{0, 1, \#, B\}$.

Define $g : A_-^* \to A_-^*$ as

$$g(w) := \begin{cases} 0 & \text{if } f_{M_{h(w)}}(w) \neq 0 \\ 1 & \text{else – that is if } f_{M_{h(w)}}(w) = 0 \text{ as } f_{M_{h(w)}} \text{ is total.} \end{cases}$$

This is a total function and we can in fact define a Turing machine M that computes it:

First M computes h and remembers w. On the result of $h(w)$ a part of M that is a universal Turing machine operates and computes $f_{M_{h(w)}}(w)$. Depending on the result, then 0 or 1 is written so that the result is $g(w)$. So $g \in S_t$.

As $h(A_\_^*) = T$, there is a $v \in A_\_^*$ with $f_{M_{h(v)}} = g$.

So $f_{M_{h(v)}}(v) = g(v) \neq f_{M_{h(v)}}(v)$ – a contradiction.

■

56 Exercise *Assume that* $\emptyset \neq L \subseteq A_\_^*$.
Prove that the following statements are equivalent definitions of non empty sets being recursively enumerable:

- *There is a (possibly partial) computable function* $\tilde{\chi}_L : A_\_^* \to A_\_^*$ *so that*

$$\tilde{\chi}_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 \text{ or undefined} & \text{else} \end{cases}$$

- *There is a (possibly partial) computable surjective function* $f : A_\_^* \to L$.

- *There is a total computable surjective function* $f : A_\_^* \to L$.

- L *is finite or there is a total computable bijective function* $f : A_\_^* \to L$.

57 Exercise *Which of the following sets is decidable?*

- $\{(n, m) \in \mathbb{N}^2 | n = \Sigma(m)\}$

- $\{u | u \in \{0,1\}^*,\ f_{M_u}(v) = v$ *for all* $v \in \{0,1\}^*\}$

- $\{u | u \in \{0,1\}^*,\ M_u$ *stops for each input*$\}$

- $\{u | u \in \{0,1\}^*,\ \{f_{M_u}(v) | v \in \{0,1,\#\}^*\}$ *is recursively enumerable* $\}$

- $\{u | u \in \{0,1\}^*,\ \{f_{M_u}(v) | v \in \{0,1,\#\}^*\}$ *is decidable* $\}$

58 Exercise $B := \{u|$ *for all* $v \in \{0,1\}^* : f_{M_u}(v) \in \{0,1\}$ *or* $f_{M_u}(v)$ *is undefined.*$\}$

- *is* B *decidable ?*

- *is B recursively enumerable ?*
- *is B^c recursively enumerable ?*
- *does there exist a decidable set $B' \subseteq \{0,1\}^*$ so that $\{f_{M_u} | u \in B\} = \{f_{M_u} | u \in B'\}$?*

59 Exercise *Let $T \subseteq \{0,1\}^*$ be a set so that $\{f_{M_u} | u \in T\}$ is equal to the set of total computable functions S_t.*
Is T reducible to the halting problem or the special halting problem?

Think about what you can say about decidability and recursive enumerability if a problem A can be reduced to a problem B and if for A or B it is already known whether it is decidable or recursively enumerable. What can you say about the complements?
One possibility would be to make a table of all combinations and consequences.

60 Exercise *Construct a problem that can not be recursively enumerated and where also the complement can not be recursively enumerated.*

5 The incompleteness of elementary number theory

Although this course is about complexity and computability, we will make a **small** digression to see how computability and provability are connected. But to be honest: one of the reasons is surely also that I find the incompleteness theorems especially beautiful. . .

5.1 Proof systems

Just like with *computation* we have to first define exactly what a *proof* is. On a conference I once attended, a sociologist said that a proof is something that is *intersubjectively verifiable.* Unfortunately she was most likely correct with this definition: a proof is a reasoning *everybody accepts.* But in fact we would like a proof to be something stronger – and most of all: something objective.
We want that all theorems can be deduced from very few axioms and fixed *rules of deduction.* Even then we would still have to *believe* in the axioms and rules of deduction, but as soon as you agree on these, a proof would be something objective.
Nevertheless it is important that even things that one would consider as *obvious* must be fixed by the proof system – like e.g. the *law of the excluded middle.* That law is used to prove that if you assume that a statement is not true – and this leads to a contradiction – the statement must be true. So if something is not not true, then it must be true and nothing in the middle.
At the moment we are somehow mixing up words like *true* and *provable* – later we will be more exact.
Also the principle of induction must be fixed in the proof system and not added later as *obvious.* Either it is a rule of deduction (and part of the system) – or not. As soon as the rules of deduction are fixed, there is nothing like *obvious.* The term obvious goes more into the direction of *intersubjectively verifiable* but in fact also often in the direction of something very subjective. . .
Especially the principle of the excluded middle caused a lot of discussion in the scientific community as it was not accepted by all mathematicians. In 1908 L.E.J. Brouwer – a famous mathematician – attacked this principle in his article *The untrustworthiness of the principles of logic.* Maybe you find this attack weird – but as Brouwer was an exceptionally good mathematician you can be sure that he had good reasons. If you search for the keyword *intuitionism* on the internet, you will find a lot about this discussion.
The question is whether for each statement A either A or $\neg A$ is valid – and nothing in between. Brouwer believed this for statements about finite sets,

but did not accept it for statements involving infinite sets.
How careful you have to be to accept assumed truths shows Cantor's example of a set in his *Beiträge zur Begründung der transfiniten Mengenlehre*:
A set is a gathering together into a whole of definite, distinct objects of our perception and of our thought which are called elements of the set.
This description – or definition(?) – seems acceptable, but nevertheless led to paradoxes like Russel's *set of all sets that don't contain themselves.*
But even as the axioms and rules of deduction are accepted, the proofs that really consist of elementary steps only, are **extremely** long and absolutely unreadable. The result is that real world proofs look quite different and that often errors are found in real world proofs and that over the correctness of some proofs the opinions differ.
Now we will define precisely what a proof system is and what a proof is. Proofs can in fact be found and verified by a computer. So one could ask what mathematicians are still needed for. Unfortunately (or fortunately?) a computer would have to go through an unbelievable amount of data and would need **much** too much time even to find simple proofs. A mathematician can use his insight and intuition to navigate much more efficiently in the amount of data and thereby find proofs a computer can never find – or to be precise: for which a computer would need **much** too much time.

78 Definition *A proof system is a 4-tuple $\mathcal{S} = (A, L, S_0, S)$ with the following properties:*

- *A is a finite alphabet not containing the symbol "$|$"*
- *$L \subseteq A_-^*$ is a decidable set – the language of $\mathcal{S}$*
- *$S_0 \subseteq L$ is a decidable set – the set of axioms of $\mathcal{S}$*
- *S is a decidable set of words of the form $w_1|w_2|\ldots|w_r$, with $w_i \in L$ for all $1 \leq i \leq r$ – the set of rules of deduction or deduction rules of $\mathcal{S}$*

If $w_1|w_2|\ldots|w_r \in S$ then we say that w_r follows directly from $w_1, \ldots, w_{r-1}$.

79 Definition *We write $A_| = A_- \cup \{|\}$.*
If $\mathcal{S} = (A, L, S_0, S)$ is a proof system then the string $w_1|w_2|\ldots|w_r \in A_|^*$ with $w_i \in L$ for $1 \leq i \leq r$ is called a proof of w_r in $\mathcal{S}$ if for all $i \leq r$ we have that $w_i \in S_0$ or there are $1 \leq i_1 \ldots i_s < i$ with $w_{i_1}|w_{i_2}|\ldots|w_{i_s}|w_i \in S$.

In this case we also say that w_r is deducible in $\mathcal{S}$.

80 Lemma *If $\mathcal{S}$ is a proof system, then the set $\{b \in A_|^* | b$ is a proof$\}$ is decidable.*

Proof: If b is of the form $w_1|w_2|\ldots|w_r$ with $w_i \in A_-^*$, one can test whether the definition is fulfilled by testing whether $w_i \in S_0$ and possibly afterward evaluating all combinations of indexes $i_1 < \cdots < i_s < i$ to see whether $w_{i_1}|w_{i_2}|\ldots|w_{i_s}|w_i \in S$. For this we use the routines that can decide S_0 and S.

■

81 Corollary *If $\mathcal{S}$ is a proof system, then the set $\{w|w$ is deducible in $\mathcal{S}\}$ is recursively enumerable.*

Proof: Assume that w_0 is an arbitrary statement that is deducible in $\mathcal{S}$ – e.g. an axiom. M is the machine that started on $b \in A_|^*$ tests whether b is a proof. In that case it prints the proven statement (the statement that is after the last $|$). In case it is not a proof, M prints w_0.

Then we have $\{w|w$ is deducible in $\mathcal{S}\} = f_M(A_|^*)$

■

5.2 An important example: the proof system Z_E

Now we will define and examine the proof system $Z_E = \{A_E, L_E, (S_0)_E, S_E\}$ of elementary number theory. For this system we will see one of the most important theorems in this course.
We have
$A_E = \{0, 1, x, (,), +, \cdot, =, \wedge, \vee, \neg, \rightarrow, \forall, \exists\}$
These are just symbols and in the proof system they have no other role than that: letters that occur in strings. But for us, a proof system that just contains meaningless strings is not really interesting. We are interested in *meaningful* statements – that is: we want to *interpret* the strings. In this interpretation these letters get the meaning that we would expect – e.g. "$\vee$" means *"or"* and "$\forall$" gets the meaning *"for all"*. The symbol "$\rightarrow$" gets the meaning *"implies"* – or in order to make it even more clear that we are talking about an interpretation: it gets the meaning *"then also"*. If the left hand side of the letter is valid *"then also"* the right hand side. But this is our interpretation and that means nothing inside the system. You can not deduce things inside the system just because the interpretation suggests that that should be OK.
In order to define the other sets we first need a bit of work:

82 Definition • *a word of the form xw with $w \in \{0,1\}^+$ is called a variable.*
$V := \{xw | w \in \{0,1\}^+\}$
The set of terms is the smallest set T for which we have

- $0 \in T$, $1 \in T$, $V \subseteq T$
- *if $a, b \in T$ then $(a \cdot b)$ and $(a + b)$ are also in T.*

Examples of terms are $x10$, $(x10 + x1)$ and $((1 + x100) \cdot (0 + x100))$.

83 Definition *The set of arithmetic predicates is the smallest set P for which we have*

- *if a and b are terms (that is: in T), then $(a = b) \in P$*
- *if A and B are in P, then $(\neg A)$, $(A \vee B)$, $(A \wedge B)$, $(A \rightarrow B)$ are also in P*
- *if $y \in V$ and $A \in P$, then $(\exists y A)$ and $(\forall y A)$ are in P*

examples of arithmetic predicates are
$(x1 = 0)$, $(\exists x11((x11 = 0) \vee (x11 = (x0 \cdot x110))))$ and $(\exists x0(\forall x10((x0 + x11) = x10)))$

84 Definition *Let y be a variable.*

- *If a and b are terms and if the variable y occurs in a or b, then y is free in $(a = b)$.*
- *If A is an arithmetic predicate in which y occurs, then y occurs bonded in $(\exists y A)$ and $(\forall y A)$.*
- *If A and B are arithmetic predicates and y occurs freely in A or B, then y occurs freely in $(A \vee B)$, $(A \wedge B)$ and in $(A \rightarrow B)$.*

  If y occurs freely in A then y occurs freely in $(\neg A), (\exists z A), (\forall z A)$.

  For bonded *this part is completely analogous.*

An arithmetic predicate without any free variables is called an arithmetic statement.

Note that in a predicate a variable can occur freely **and** bonded at different places.

Now we can define $L_E := \{A | A \text{ is an arithmetic predicate}\}$
Of course we have to prove:

85 Lemma *L_E is decidable.*

But as the definitions can be *easily* tested by a Turing machine, that is relatively obvious.

Until now we have only spoken about terms like *provability* or *deducibility*, but what we have proven or deduced are just strings – which for themselves have no meaning at all! It is us who **interpret** these strings. For example the statement (string) $(0 = 1)$ can be deducible in some proof systems. One can e.g. choose it as an axiom.
I guess we would not choose it as we consider it to be *not true*, but until now the concepts of *true* or *false* have not yet been introduced!
We first have to define what we mean when we say that an arithmetic statement is *true*.

86 Definition *A mapping $\phi : V \to \mathbb{N}$ is called an $\underline{\text{assignment}}$ of the variables. The $\underline{\text{value}}$ of a term t for an assignment ϕ is defined as*
$\phi(0) = 0$, $\phi(1) = 1$, $\phi((a + b)) = \phi(a) + \phi(b)$, $\phi((a \cdot b)) = \phi(a) \cdot \phi(b)$

15 Example *Assume $t = ((x0 + x1) \cdot (x0 + 1))$ and an assignment ϕ is given with $\phi(x0) = 1$, $\phi(x1) = 2$. Then $\phi(t) = \phi(x0 + x1) \cdot \phi(x0 + 1) = (\phi(x0) + \phi(x1)) \cdot (\phi(x0) + \phi(1)) = (1 + 2) \cdot (1 + 1) = 6$*

Notation

- for a predicate A, a variable x and a term t we write $A|_{x=t}$ for the predicate one gets if in each place where x occurs freely, it is replaced by t.
- for a predicate A in which $x_1, \ldots, x_n$ occur freely, we write $A(x_1, \ldots, x_n)$ and for $A(x_1, \ldots, x_n)|_{x_1=t_1,\ldots,x_n=t_n}$ also $A(t_1, \ldots, t_n)$.

  The replacement of the variables is of course simultaneous and not one after the other. Note that e.g. t_1 could contain a variable – e.g. x_2 that is also replaced. So if t_1 would be inserted first and then t_2, then the x_2 in t_1 would also be replaced – which is of course not intended.
- the special term $(1+(1+(\cdots+1)\ldots)$ ($n \geq 1$ times) is also abbreviated as $\bar{n}$. In case $n = 0$, $\bar{n}$ denotes the term 0.

87 Definition *Assume that a, b are terms and that A, B are predicates without free variables. Furthermore let x be a variable and C an arbitrary predicate (in which x can occur or not).*

- *If x occurs freely in C, then $C(x)$ is* <u>*true*</u> *if $C(\bar{n})$ is true for all $n \in \mathbb{N}$.*

  For the remaining cases assume that there are no free variables:

- *$a = b$ is true, if $\phi(a) = \phi(b)$ for all assignments $\phi()$.*
- *$A \wedge B$ is true, if A* **and** *B are true.*
- *$A \vee B$ is true, if A* **or** *B* **or** *both are true.*
- *$\neg A$ is true, if A is not true.*
- *$A \to B$ is true if $(B \vee (\neg A))$ is true.*
- *If $\exists x C$ does not contain free variables, $\exists x C$ is true if there is a number $n \in \mathbb{N}$ so that $C|_{x=\bar{n}}$ is true.*
- *If $\forall x C$ does not contain free variables, $\forall x C$ is true, if $C|_{x=\bar{n}}$ is true for all $n \in \mathbb{N}$.*

For not true *we also say* <u>*false*</u>.

16 Example • *The predicate*

$((x0 + x1) \cdot (x0 + x1)) = ((x0 \cdot x0) + (((1 + 1) \cdot (x0 \cdot x1)) + (x1 \cdot x1)))$ *is true. (Binomial formula $(a + b)^2 = a^2 + 2ab + b^2$)*

- *The predicate*

  $(\exists x1((x0 + x1) = x0))$ *is true.*

- *The predicate*

  $(\forall x1((x0 + x1) = x0))$ *is not true.*

Of course we want to investigate the connection between *true* and *deducible*. A proof system where also false statements can be deduced is not what we want. So we want axioms that are true and rules of deduction that preserve truth.

88 Definition *If t is a term, y a variable and $A(y)$ a predicate, then y is called* <u>*free for*</u> *t in $A(y)$ if all variables that occur in t, occur only freely or not at all in $A(t)$.*

17 Example *In the predicate* $((\exists x0(((1+1)+x0)=x1))\rightarrow(\exists x0((1+x0)=x1)))$ *the variable* $x1$ *occurs freely – so it can be written as* $A(x1)$.
In this predicate $x1$ *is free for (e.g.) the terms* $x11$, $x10$, 1, $((1+1)+1)$. *For terms that contain* $x0$ *– like* $x0$ *or* $((1+1)+x0)$ *–* x_1 *is not free.*
If y *is not free for a term* t *in* $A(y)$, *then a true statement* $A(y)$ *can become a false statement* $A(t)$. *For example:* $A(x1)$ *is true (check that!) but* $A((1+1)+x0)$ *is false (check that too!).*

61 Exercise *Given the predicates*

- $((\exists x1(\forall x0((1+x1)=(1+x0))))\rightarrow(\forall x1(\exists x0((1+x1)=(1+x0)))))$
- $((\exists x1((x0+x1)=\bar{n}))\rightarrow(\forall x1((x0+\bar{n})=x1)))$

Check whether the predicates are true or false and whether the variable $x0$ *in the statements is free for* $(1+1)$, $\bar{n}$, $(x0+\bar{n})$ *or* $(x0+x1)$.

62 Exercise *Give predicates that have the following meaning:*

- *The constant number* i *is a prime number (of course in the predicate* $\bar{i}$ *is used).*
- *There is exactly one number that is divisible by 2.*
- *There are infinitely many prime twins – that is pairs* $(n, n+2)$ *so that* n *and* $n+2$ *are prime numbers.*

Now we will fix our axioms and rules of deduction.
The set $(S_0)_E$ of axioms is defined as follows:

89 Definition *First we will give the axioms for logical combinations.*
Let A, B, C *be arithmetic predicates. Then the following are axioms:*

1.) $(A\rightarrow(B\rightarrow A))$

2.) $((A\rightarrow B)\rightarrow((A\rightarrow(B\rightarrow C))\rightarrow(A\rightarrow C)))$

3.) $(A\rightarrow(B\rightarrow(B\wedge A)))$

4.) $((A\wedge B)\rightarrow A)$

5.) $((A\wedge B)\rightarrow B)$

6.) $((A\rightarrow(A\vee B)))$

7.) $((B \rightarrow (A \vee B)))$

8.) $((A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C)))$

9.) $((A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A))$

10.) $((\neg(\neg A)) \rightarrow A)$

Now the axioms for quantifiers:
Assume that y is a variable and that $A(y)$ is a predicate in which y occurs freely. Furthermore let t be a term for which y is free in $A(y)$. Then the following are axioms:

11.) $((\forall y A(y)) \rightarrow A(t))$

12.) $(A(t) \rightarrow (\exists y A(y)))$

And finally the axioms for the natural numbers. Let y be a variable, $A(y)$ a predicate in which y occurs freely and a, b, c terms. Then the following are axioms:

13.) $((A(0) \wedge (\forall y (A(y) \rightarrow A(y+1)))) \rightarrow A(y))$ *(axiom of induction)*

14.) $(((a+1) = (b+1)) \rightarrow (a = b))$

15.) $(\neg((a+1) = 0))$

16.) $((a = b) \rightarrow ((a = c) \rightarrow (b = c)))$

17.) $((a = b) \rightarrow ((a+1) = (b+1)))$

18.) $((a+0) = a)$ *(0 the neutral element of summation)*

19.) $((a + (b+1)) = ((a+b)+1))$ *(associativity of summation)*

20.) $((a \cdot 0) = 0)$

21.) $((a \cdot (b+1)) = ((a \cdot b) + a))$ *(distributivity)*

The set $(S_0)_E$ of axioms of elementary number theory is the set of all strings with letters from A_E^* that are described by one of the axioms 1.) to 21.).

The principle of proving is that you have basic **true** axioms and rules of deduction that preserve truth. So we should check whether all our axioms are really true – according to our own definition of truth. We will just give an example, but you should check some other axioms too!
1.) $(A \to (B \to A))$ is true if (and only if) $(\neg A \vee (B \to A))$ is true and that again if $(\neg A \vee (\neg B \vee A))$ is true. But this is true if $\neg A$ or $\neg B$ or A are true – so always. So this axiom is in fact true for all predicates A and B.
One more example:
12.) $(A(t){\to}(\exists y A(y)))$ is true if $(\neg A(t) \vee (\exists y A(y)))$ is true. That is true if $\neg A(t)$ is true or $\exists y A(y)$ is true. If $\exists y A(y)$ is true, that is OK, so let us assume that it is false. Then there does not exist a number $n \in \mathbb{N}$ so that $A|_{y=\bar{n}}$ is true. Then $\neg A|_{y=\bar{n}}$ is true for each number $n \in \mathbb{N}$. Each assignment of variables gives a value – $\bar{n}$ – for the term t for which $\neg A(\bar{n})$ is true, so $\neg A(t)$ is true and therefore also $(A(t){\to}(\exists y A(y)))$. So this axiom is also true for all predicates $A(t), A(y)$.
Analogously you can also work out the other axioms.

Now we will define the set S_E of rules of deduction:

90 Definition *Let $A, B, A(y), C$ be arithmetic predicates and y a variable so that y occurs freely in $A(y)$ and if it occurs in C, it does not occur freely in C. Then the following are rules of deduction:*

a.) $A|(A{\to}B)|B$

b.) $(C{\to}A(y))|(C{\to}(\forall y A(y)))$

c.) $(A(y){\to}C)|((\exists y A(y)){\to}C)$

The set of deduction rules of elementary number theory is the set of all strings with letters from $(A_E)^*_|$ described by the rules a.),b.),c.).

We will just show that rule c.) preserves truth – the rest can be done analogously: If $(A(y){\to}C)$ is true, then C is true or $\neg A(y)$. If C is true, then also $((\exists y A(y)){\to}C)$. If $\neg A(y)$ is true, then $\neg A|_{y=\bar{n}}$ is true for all $n \in \mathbb{N}$. So $A|_{y=\bar{n}}$ is false for all $n \in \mathbb{N}$ and no $n \in \mathbb{N}$ exists so that $A|_{y=\bar{n}}$ is true. This means that $\exists y A(y)$ is false and again $((\exists y A(y)){\to}C)$ is true. So this rule preserves truth.

91 Remark *It is* in principle *easy to check whether a string fulfills one of the rules 1.) to 21.) for axioms or one of the rules a.),b.),c.) for deduction rules. So S_E and $(S_0)_E$ are decidable.*

But note *that when it comes to using them in a formal proof, these axioms and strings obtained by deduction are not subject to interpretation. They are strings and nothing more!*
E.g. in
$((A(0) \wedge (\forall y(A(y) \rightarrow A(y+1)))) \rightarrow A(y))$
the $\forall y$ *does not mean that something has to be done* for all y – *e.g. to the string* $(A(y) \rightarrow A(y+1))$. *This axiom is just a string fulfilling certain requirements: the* $\forall$ *is at the right place, each* $\rightarrow$ *is where it is allowed according to the rules, etc.*

The following example is from *S.C. Kleene: Introduction to Metamathematics* and was also cited in the book from W. Paul. It shows how complicated it can be to prove even the simplest statements if one has to do each step according to this formalism in Z_e.
The predicate we want to prove is $(x0 = x0)$. This is just a string for which a sequence of deduction steps has to be found that proves it. You may be tempted to say that that *is obviously true* – but that would refer to the truth of the predicate and not to the fact whether it is deducible in the system!
In order to make it better understandable we also add numbers to the steps and some explanations. They are of course not part of the proof. The proof is only the middle part.

(1)	$((x0 = x1)\to((x0 = x10)\to(x1 = x10)))\vert$	axiom 16
(2)	$((0 = 0)\to((0 = 0)\to(0 = 0)))\vert$	axiom 1
(3)	$(((x0 = x1)\to((x0 = x10)\to(x1 = x10)))\to$ $(((0 = 0)\to((0 = 0)\to(0 = 0)))\to$ $((x0 = x1)\to((x0 = x10)\to(x1 = x10)))))\vert$	axiom 1
(4)	$(((0 = 0)\to((0 = 0)\to(0 = 0)))\to$ $((x0 = x1)\to((x0 = x10)\to(x1 = x10))))\vert$	deduction rule a applied to (1),(3)
(5)	$(((0 = 0)\to((0 = 0)\to(0 = 0)))\to$ $(\forall x10((x0 = x1)\to((x0 = x10)\to(x1 = x10)))))\vert$	deduction rule b applied to (4)
(6)	$(((0 = 0)\to((0 = 0)\to(0 = 0)))\to$ $(\forall x1(\forall x10((x0 = x1)\to((x0 = x10)\to(x1 = x10))))))\vert$	deduction rule b applied to (5)
(7)	$(((0 = 0)\to((0 = 0)\to(0 = 0)))\to$ $(\forall x0(\forall x1(\forall x10((x0 = x1)\to((x0 = x10)\to(x1 = x10)))))))\vert$	deduction rule b applied to (6)
(8)	$(\forall x0(\forall x1(\forall x10((x0 = x1)\to((x0 = x10)\to(x1 = x10))))))\vert$	deduction rule a applied to (2),(7)
(9)	$((\forall x0(\forall x1(\forall x10((x0 = x1)\to((x0 = x10)\to(x1 = x10))))))\to$ $(\forall x1(\forall x10(((x0 + 0) = x1)\to(((x0 + 0) = x10)\to(x1 = x10))))))\vert$	axiom 11 $t = (x0 + 0)$ for $x0$
(10)	$(\forall x1(\forall x10(((x0 + 0) = x1)\to(((x0 + 0) = x10)\to(x1 = x10)))))\vert$	deduction rule a applied to (8),(9)
(11)	$((\forall x1(\forall x10(((x0 + 0) = x1)\to(((x0 + 0) = x10)\to(x1 = x10)))))\to$ $(\forall x10(((x0 + 0) = x0)\to(((x0 + 0) = x10)\to(x0 = x10)))))\vert$	axiom 11 $t = x0$ for $x1$
(12)	$(\forall x10(((x0 + 0) = x0)\to(((x0 + 0) = x10)\to(x0 = x10))))\vert$	deduction rule a applied to (10),(11)
(13)	$((\forall x10(((x0 + 0) = x0)\to(((x0 + 0) = x10)\to(x0 = x10))))\to$ $(((x0 + 0) = x0)\to(((x0 + 0) = x0)\to(x0 = x0))))\vert$	axiom 11 $t = x0$ for $x10$
(14)	$(((x0 + 0) = x0)\to(((x0 + 0) = x0)\to(x0 = x0)))\vert$	deduction rule a applied to (12),(13)
(15)	$((x0 + 0) = x0)\vert$	axiom 18
(16)	$(((x0 + 0) = x0)\to(x0 = x0))\vert$	deduction rule a applied to (15),(14)
(17)	$(x0 = x0)$	deduction rule a applied to (15),(16)

63 Exercise *Prove in Z_E: $((A \wedge B)\to(B \wedge A))$*

Although it is obviously very complicated to prove even trivial theorems in this system, all theorems known in elementary number theory can be proven in this system. The principle is like for Turing machines, where it is also difficult to describe simple machines in every detail, but as soon as you have larger *building blocks'* that can be assembled, also complicated machines can *in principle* be built, resp. complicated theorems can *in principle* be deduced in the system.

64 Exercise *Give a proof of $(x1 = x1)$ using the already proven $(x0 = x0)$. Using $(x0 = x0)$ makes a much shorter derivation possible.*

Now we will define some properties of proof systems that we would consider necessary or at least useful:

92 Definition *Let $\mathcal{S} = (A, L, S_0, S)$ be a proof system with the symbol $\neg \in A$. We* **interpret** *the symbol „$\neg$"always as "***not***", but that is only important for our intuition and makes no principal difference.*
A truth definition is a total function $L \to \{true, false\}$ so that for each element $w \in L$ **exactly one** *of $w, \neg w \in L$ is mapped to* true *(and the other to false). For the element mapped to* true *(resp.* false*) we also say that it* is true *(resp.* is false*).*

- *$\mathcal{S}$ is called consistent if for no $w \in L$ both w and $\neg w$ are deducible in $\mathcal{S}$.*
- *$\mathcal{S}$ is called complete if for each $w \in L$ either w or $\neg w$ or both are deducible.*
- *if there is a truth definition for $\mathcal{S}$, then $\mathcal{S}$ is called correct (with respect to this truth definition) if all deducible statements are true.*

So it would be ideal if Z_E would be a correct and complete proof system, but note that we just *believe* that our definition of truth fulfills the requirement of a truth definition. We did not prove that...

93 Remark *Correct proof systems are consistent.*

Proof: As only one of $w, \neg w$ can be true – due to the properties of a truth definition, in a correct proof system at most one of the two can be deduced.

■

As a warning one should again think of the *"proof system"* that allows Russel's paradox of *the set of all sets that do not contain themselves.* In that system it could be deduced that the set contains itself and that it does not contain itself. So the system was not consistent and therefore also not correct.

65 Exercise *Prove: Z_E is not consistent if and only if* **all** *predicates in L_E are deducible in Z_E.*

94 Lemma *If $\mathcal{S} = (A, L, S_0, S)$ is a correct and complete proof system with truth definition, then the set $\{w | w \in L, w$ true$\}$ of true statements is decidable.*

Proof: For an input $w \in A_{\_}^{*}$ we first test whether $w \in L$. If that is not the case, we write 0. If $w \in L$, all strings from $A_{|}^{*}$ are enumerated in increasing length and lexicographic order. For each string b it is tested whether b is a proof of w or of $\neg w$. If a proof of w is found the machine writes 1 and stops, in case a proof of $\neg w$ is found, it writes 0 and stops. As $\mathcal{S}$ is complete, the machine will stop sooner or later. So a proof of w is found if and only if w is true and a proof of $\neg w$ is found if and only if $\neg w$ is true – and that means: if w is false.

■

To get an intuition on how realistic this way of finding proofs is, assume that a computer can generate 1 000 000 000 strings per second and test them to determine whether they are a proof of a given statement. Then a computer that works as in the proof of Lemma 94 would need approximately $6.4 * 10^{980}$ times the age of the universe to find the proof of $(x0 = x0)$ we have given.
As statements about Turing machines can be expressed in the language of a proof system, we can apply our results for computable functions:

95 Definition *Let $\mathcal{S} = (A, L, S_0, S)$ be a proof system with truth definition. Assume that there is a total computable function $H : \{0, 1, \#\}^* \to L$ so that for all $u, v \in \{0, 1\}^*$ we have:*
$H(u\#v)$ is true if and only if the 1-tape Turing machine M_u started on v stops. Then we say that $\mathcal{S}$ contains the halting problem.

Or with other words: We say that $\mathcal{S}$ contains the halting problem if the halting problem is reducible to the set of true statements.

96 Theorem *If $\mathcal{S} = (A, L, S_0, S)$ is a proof system that contains the halting problem, then the set $W := \{h \in L | h$ true$\}$ of true statements is not decidable.*

Proof: The condition says that the halting problem is reducible to W. If W was decidable, the halting problem would be decidable too.

■

97 Corollary *If a proof system $\mathcal{S} = (A, L, S_0, S)$ contains the halting problem and $\mathcal{S}$ is correct, then $\mathcal{S}$ is incomplete.*

Proof: This corollary follows immediately from Theorem 96 and Lemma 94.

■

98 Theorem *Z_E contains the halting problem.*

Proof: The approach is similar to the one in the proof of Theorem 52. The idea is: We code configurations and computations as numbers. Then there is a finite computation of a given Turing machine started on a given input if and only if there is a number coding a finite computation of that Turing machine started on that input. We will show that the test whether a number codes such a computation can be described by means of arithmetic predicates. Or a bit more precisely:

Given a Turing machine M with Gödelisation u and input v. We will now describe what $H(u\#v)$ is. Your experience with Turing machines is surely sufficient to be convinced that $H(u\#v)$ can be computed from $u\#v$.

If $M = (Z, A, \delta, z_0, E)$ and $/ \notin (Z \cup A)$ then M started on v stops if and only if there is a word $w = /c_0/\dots/c_t/$ so that

1.) c_0 has the form $B \dots B z_0 v B \dots B$

2.) $c_i \vdash c_{i+1}$ for $0 \le i \le t-1$

3.) $|c_i| = |c_j|$ for $0 \le i, j \le t$

4.) no c_i contains a symbol describing a state at its boundary

5.) c_t is a stop configuration – that means: contains a symbol from E

Again 3.) and 4.) are present just for technical reasons – that is to make some things a bit easier to write up and have less cases.

The coding of a string as a number is done similar to the proof of Theorem 52 only that here we allow the empty word, have to code the letter „/“ and use a prime number as base. Or precisely:

Let p be a prime number with $p \ge |Z|+|A|+2$ and let $\psi_0 : Z \cup A \cup \{/\} \to \{1, \dots, p-1\}$ be an injective function. For $a_1 \dots a_k \in (Z \cup A \cup \{/\})^*$ we define $\psi : (Z \cup A \cup \{/\})^* \to \mathbb{N}$ as

$\psi(a_1 \dots a_k) = \sum_{i=1}^{k} \psi_0(a_i) p^{k-i}$.

The empty word is mapped to 0. For $a \in Z \cup A \cup \{/\}$ we write instead of $\overline{\psi(a)}$ also shorter $\hat{a}$.

The concatenation of strings is again an important part – so we will first discuss that:

Remark:

There is an arithmetic predicate $concat(z, x, y)$ – written shortly as „$z = [x, y]$“, so that for all strings $X, Y \in (Z \cup A \cup \{/\})^*$ we have:

If $x = \psi(X)$ and $y = \psi(Y)$, then $z = [x, y]$ is true if and only if $z = \psi(XY)$.

Proof of the remark:

For $y \in \mathbb{N}$ let $\psi^{-1}(y)$ be the string encoded by y and $L(y)$ its length (if such a string exists).

Now we will construct the desired predicate in small steps. On the right hand side of „$\equiv$“ there will always be an a bit more formal version than on the left hand side – until we have finally formulated the whole predicate in Z_E. Not yet formal parts are written in bold letters.

$z = [x, y] \equiv (((y = 0) \wedge (z = x)) \vee ((\neg(y = 0)) \wedge (\exists u((z = ((x \cdot u) + y)) \wedge (\mathbf{u} = \mathbf{p}^{\mathbf{L(y)}})))))$

$(\mathbf{u} = \mathbf{p}^{\mathbf{L(y)}}) \equiv ((\mathbf{u\ is\ a\ power\ of\ p}) \wedge (\mathbf{y} < \mathbf{u}) \wedge (\mathbf{u} \leq \mathbf{p} \cdot \mathbf{y}))$

Note that according to the definition we have $p^{L(y)} > y \geq p^{L(y)-1}$.

$(\mathbf{u\ is\ a\ power\ of\ p}) \equiv (\forall r(\forall s(((r \cdot s) = u) \rightarrow ((r = 1) \vee (\mathbf{p\ divides\ r})))))$

Here we use that p is a prime.

$(\mathbf{p\ divides\ r}) \equiv (\exists t(r = (\bar{p} \cdot t)))$

$(\mathbf{y} < \mathbf{u}) \equiv (\exists v((u = (y + v)) \wedge (\neg(v = 0))))$

$(\mathbf{u} \leq \mathbf{p} \cdot \mathbf{y}) \equiv (\exists v((u + v) = (\bar{p} \cdot y)))$

Inserting the formalized parts we get the desired predicate.

(end of the proof of the remark)

By interlocking we can now write words after each other:

$(\mathbf{u} = [\mathbf{x}, \mathbf{y}, \mathbf{z}]) \equiv (\exists w((w = [x, y]) \wedge (u = [w, z])))$, etc.

Now we can formulate the properties (1) to (5) as an arithmetic predicate. The variable w always stands for the number coding the whole string:

$(\mathbf{1}) \equiv (\exists k(\exists m(\exists a(\exists b(\exists c((w = [\hat{/}, k, \hat{/}, m]) \wedge (k = [a, b, c]) \wedge (\mathbf{a} = \psi(\mathbf{B} \ldots \mathbf{B})) \wedge (\mathbf{b} = \psi(\mathbf{z_0 v})) \wedge (\mathbf{c} = \psi(\mathbf{B} \ldots \mathbf{B}))))))))$

The k represents the first configuration and the m the remaining configurations and separation symbols.

$(\mathbf{a} = \psi(\mathbf{B} \ldots \mathbf{B})) \equiv (\forall d((\textbf{d codes a single letter in } \boldsymbol{\psi^{-1}}(\mathbf{a}))\rightarrow(d = \hat{B})))$

$(\textbf{d codes a single letter in } \boldsymbol{\psi^{-1}}(\mathbf{a})) \equiv ((d < \bar{p})\wedge(\exists e(\exists f(a = [e, d, f]))))$

If $v = v_1 \ldots v_n$ then we have

$(\mathbf{b} = \psi(\mathbf{z_0 v})) \equiv (\exists y_0(\exists y_1(\ldots \exists y_n((y_0 = \hat{z}_0) \wedge (y_1 = ((\bar{p} \cdot y_0) + \hat{v}_1)) \wedge \cdots \wedge (y_n = ((\bar{p} \cdot y_{n-1}) + \hat{v}_n)) \wedge (b = y_n)) \ldots)))$

$((\mathbf{2}) \wedge (\mathbf{3})) \equiv (\forall r(\forall x(\forall y(\forall z(((w = [r, \hat{/}, x, \hat{/}, y, \hat{/}, z])\wedge(/\ \textbf{ does not occur in x})\wedge (/\ \textbf{ does not occur in y}))\rightarrow(\mathbf{x} \vdash \mathbf{y}) \wedge (|\mathbf{x}| = |\mathbf{y}|))))))$

$(/\ \textbf{ does not occur in x}) \equiv$
$(\forall d((d \text{ codes a single letter in } \psi^{-1}(x))\rightarrow(\neg(d = \hat{/}))))$

$((\mathbf{x} \vdash \mathbf{y}) \wedge (|\mathbf{x}| = |\mathbf{y}|)) \equiv (\exists u(\exists v((\bigvee_{\delta(z,a)=(z',b,N)}((x = [u, \hat{z}, \hat{a}, v]) \wedge (y = [u, \hat{z}', \hat{b}, v]))) \vee (\bigvee_{\delta(z,a)=(z',b,R)}((x = [u, \hat{z}, \hat{a}, v]) \wedge (y = [u, \hat{b}, \hat{z}', v]))) \vee (\bigvee_{\delta(z,a)=(z',b,L)}(\bigvee_{c \in A}((x = [u, \hat{c}, \hat{z}, \hat{a}, v]) \wedge (y = [u, \hat{z}', \hat{c}, \hat{b}, v])))))))$

with the indexes of the $\bigvee$ containing all transition rules that can be applied here (that is: $z, z' \in Z, a, b \in A$).

$(\mathbf{4}) \equiv (\forall x(\forall y(\forall z(\forall d(((w = [x, \hat{/}, y, \hat{/}, z]) \wedge (/ \text{ does not occur in } y) \wedge (d \text{ codes a single letter in } \psi^{-1}(y))\wedge(\bigvee_{q \in Z}(d = \hat{q})))\rightarrow(\neg(\exists e((y = [d, e])\vee (y = [e, d])))))))))$

$(\mathbf{5})$ (somehow) $\equiv (\exists u(\exists v(\bigvee_{e \in E}(w = [u, \hat{e}, v]))))$

Although this condition just says that a symbol belonging to a stop state must occur somewhere in the whole string, the combination with the other conditions (most important (2)) guarantees that it is in c_t and that c_t is a configuration.

If we now replace the variables $a, b, v \ldots$ that were chosen as they are a bit easier to read by the variables as they should be – that is $x1, x10, \ldots$ etc. – then

$H(u\#v) = \exists w((1) \wedge (2) \wedge (3) \wedge (4) \wedge (5))$

is a statement in Z_E that is true if and only if there is a finite computation of M (with Gödelisation u) started on v.

■

66 Exercise *Can you formulate the predicate* $(\mathbf{b} = \psi(\mathbf{z_0 v}))$ *from the previous proof also without using any quantifyers?*

99 Corollary *If Z_E is correct, then Z_E is incomplete.*

100 Definition *A proof system $\mathcal{S}' = (A', L', S'_0, S')$ contains a proof system $\mathcal{S} = (A, L, S_0, S)$, if $A' \supseteq A$, $L' \supseteq L$, $S'_0 \supseteq S_0$ and $S' \supseteq S$.*
If for both systems a truth definition is given, we also require that these definitions are the same on $L \subseteq L'$.

101 Remark *If a proof system $\mathcal{S}$ contains the halting problem and if $\mathcal{S}'$ contains $\mathcal{S}$, then $\mathcal{S}'$ contains the halting problem too.*
This implies that if the larger system $\mathcal{S}'$ is correct, it is also incomplete.

As Z_E contains the halting problem, any hope to have a complete and correct proof system is lost. It is not only so that the elementary system Z_E does not have these properties – if Z_e is correct, no larger and more powerful system containing Z_e is correct and complete. So adding more axioms or deduction rules to Z_e will not make the system complete – unless we destroy its correctness.
The only possibility to get a complete and correct system is by reducing the power of Z_e. This way it can be correct and complete – but at the cost that even some elementary statements can not be expressed in the system any more.
Until now we have just seen that there are predicates that cannot be proven – but just like for non-computable functions we would like to see examples of them to judge whether there are also interesting predicates that cannot be proven. In fact the predicate that some Turing machine doesn't stop can be considered as an interesting predicate, but more for computer science than mathematics. We will now construct some predicates that are extremely interesting from the mathematical point of view and also from the general scientific (and philosophical) point of view.
First:

102 Lemma *Let H be as in the proof of Theorem 98. If $u, v \in \{0,1\}^*$ and M_u started on v stops, then $H(u\#v)$ is deducible in Z_E.*

Proof: (very rough sketch)

A finite computation of M_u started on v is of course a proof (outside of Z_E and not in the formal meaning of the word), that M_u started on v stops. In fact *somehow* the best proof ever…

As the fact that M_u started on v stops is equivalent to $H(u\#v)$ such a finite computation also proves $H(u\#v)$. In fact such a finite computation can even be used to construct the number w with the properties (1),…,(5) for $H(u\#v)$.

Techniques as in the proof of Theorem 98 where we construct the H for Z_E, make it possible to translate a formal finite computation (that is: a sequence of configurations describing a computation with a stop configuration as the last one) to a real proof of $H(u\#v)$ in Z_E. Unfortunately the details of this are neither easy nor short.

■

103 Theorem *(Gödel's first incompleteness theorem)*
If $\mathcal{S} = (A, L, S_0, S)$ is a correct proof system containing the halting problem, one can explicitly construct a statement that can not be proven in $\mathcal{S}$ and where also the opposite statement (the negation) can not be proven in $\mathcal{S}$. Outside of the system $\mathcal{S}$ one can reason that the statement we will construct in the proof is wrong.

Proof: We use the diagonal argument again:

Let H be the function of Definition 95. The machine Q works as follows: started on $w \in \{0,1\}^*$ it computes $H(w\#w)$ and then $\neg H(w\#w)$. Then all strings in $A_|^*$ are enumerated with increasing length and in lexicographic order for strings of the same length. For each string it is tested whether it is a proof of $\neg H(w\#w)$. If that is the case, Q stops, otherwise it continues. So Q stops if and only if there is a proof of $\neg H(w\#w)$.

Let q be a Gödelisation of Q.

If $H(q\#q)$ can be deduced, this means that $H(q\#q)$ is true as $\mathcal{S}$ is correct. So Q started on q stops, which again means that a proof of $\neg H(q\#q)$ is found. So $H(q\#q)$ is true and $\neg H(q\#q)$ can be proven in the system, which is a contradiction to $\mathcal{S}$ being correct.

If $\neg H(q\#q)$ can be deduced, this means that M_q and Q started on q don't stop as $\mathcal{S}$ is correct. But on the other hand our assumption says that there is a proof of $\neg H(q\#q)$, so Q finds the proof of $\neg H(q\#q)$ – and stops. This is again a contradiction to $\mathcal{S}$ being correct.

So neither $\neg H(q\#q)$ nor $H(q\#q)$ can be deduced in the system.

If $H(q\#q)$ would be true, if $\mathcal{S}$ contains Z_E, we can use Lemma 102 and conclude that there would be a proof of $H(q\#q)$, while we have shown that there isn't. So $H(q\#q)$ has to be false.

But even without Lemma 102: if $H(q\#q)$ would be true, Q started on q would stop, as $H()$ reduces the halting problem to the set of true statements. So it would find a proof of $\neg H(q\#q)$ in $\mathcal{S}$ although we have shown that such a proof can not exist.

■

The term *consistent* seems to be easier to work with, as we don't need a truth definition for it. The question is only whether there are words P for which P and $\neg P$ can be deduced. So we would at least want to prove that a system is consistent!
In order to prove that in the system the first question is whether you can say that within the system – that is whether a word in the language of the system has that *meaning*. The meaning of the words is not part of the system, but is something we attach to the words from the outside and there are normally infinitely many words in the system that have the same meaning.
In the following examples our interpretation depends for example on the choice of the function H mapping the halting problem to L.
In the next proofs we will use an idea that can be used in a very general context. If you have something that can be found by a Turing machine, then the statement that it exists (or does not exist) can be expressed in (e.g.) Z_e by saying that the Turing machine that searches for it stops or does not stop. We can e.g. say that Z_e contains a statement meaning *the 4-colour theorem is true* by convincing ourself that a Turing machine can list all plane graphs and test them for colourability. So we can write a Turing machine that when started on 0 stops if and only if there is a counterexample. If the Gödelisation of this Turing machine is u then $\neg H(u\#0)$ says that no counterexample exists – so it means that the 4-colour theorem is true.
We will now **very shortly** discuss consistency of a proof system and which statements can be formulated in the system.

104 Lemma *If $\mathcal{S} = (A, L, S_0, S)$ is a proof system containing the halting problem, then there is a word in L that means „$\mathcal{S}$* is consistent*“.*

Proof: Let C be a Turing machine that works as follows: started on the string 0 it generates all strings with symbols from $A_|$ and tests whether they are a proof. If that is the case, the proven statement w is added to a list (e.g. on another track) and it is tested whether the list also contains $\neg w$ or whether w is in fact of the form $\neg w'$ for some w' already in the list. If this is the case, C stops. If C never finds such statements, it continues infinitely long.

So C stops if and only if $\mathcal{S}$ is not consistent. So the statement „*C does not stop*“ means „*$\mathcal{S}$ is consistent*“.

But the statement „*C does not stop*“ can be formulated in $\mathcal{S}$: if c is a Gödelisation of C and H as in Definition 95, then it is the statement $\neg H(c\#0)$.

From now on, for $\neg H(c\#0)$ we will write shorter Consis($\mathcal{S}$).

■

105 Lemma *If $\mathcal{S} = (A, L, S_0, S)$ is a proof system containing the halting problem and $w \in L$, then there is a word dedu(w) in L that means „w* can be deduced *in $\mathcal{S}$“.*

Proof: Let M be the machine that started on w enumerates all proofs. If a proof of w is found it stops as soon as that is found, otherwise it continues searching forever. So M stops if and only if w is deducible in $\mathcal{S}$. If m is a Gödelisation of M, then the word meaning „*w can be deduced in $\mathcal{S}$*“ is $H(m\#w)$.

■

106 Lemma *If $\mathcal{S} = (A, L, S_0, S)$ is a proof system containing Z_E and $w \in L$ is deducible in $\mathcal{S}$, then dedu(w) is deducible in Z_E and therefore also in $\mathcal{S}$.*

Proof: This follows from Lemma 102.

■

107 Theorem *(Gödel's second incompleteness theorem)*
If $\mathcal{S} = (A, L, S_0, S)$ is a consistent proof system containing Z_E, then Consis($\mathcal{S}$) is not deducible in $\mathcal{S}$.

Proof: A citation in Cohen's book *Computability and Logic* that refers to this proof says:

Though the idea is reasonably understandable, the details are exceptionally long and technical, and are beyond the scope of texts much more advanced than this one.

And therefore they are surely *beyond the scope* of a course that focuses on complexity and computability and where the incompleteness theorems are just some – hopefully interesting – digression.

All we can give here is the idea of the proof:

We start again with the machine Q of the first incompleteness theorem (Theorem 103) and its Gödelisation q.

First it is proven that

1.) if $\mathcal{S}$ is consistent, then $\neg H(q\#q)$ is not deducible in $\mathcal{S}$.

If $\neg H(q\#q)$ would be deducible in $\mathcal{S}$, Q started on q would find a proof and would stop. But according to Lemma 102 this means that then there is a proof of $H(q\#q)$ in Z_E and therefore $\mathcal{S}$ would not be consistent (a contradiction).

This proof can be formalized in Z_E as a proof for

$1.)_{Z_E}$ $\mathrm{Consis}(\mathcal{S}) \rightarrow \neg(\mathrm{dedu}(\neg H(q\#q)))$

(but this is something we will definitely not do...)

If Consis($\mathcal{S}$) would be deducible in $\mathcal{S}$, according to deduction rule a.) of Z_E (that must also be present in $\mathcal{S}$ containing Z_E) also $\neg(\mathrm{dedu}(\neg H(q\#q)))$ is deducible in $\mathcal{S}$.

This is even provable in Z_E (and not only outside), so $\neg H(q\#q)$ is not deducible.

According to the definition of Q this means that Q does not stop and therefore $\neg H(q\#q)$. This again can be concluded in Z_E (which again we will just state and not prove here) and so according to Lemma 106 also $\mathrm{dedu}(\neg H(q\#q))$.

This means that in $\mathcal{S}$ we can prove both: $\mathrm{dedu}(\neg H(q\#q))$ and $\neg(\mathrm{dedu}(\neg H(q\#q)))$. But this is a contradiction to $\mathcal{S}$ being consistent.

■

In a very vague and informal way one can state this theorem as *If inside your system you can prove that the system is consistent, then it proves that it is not!* Of course you still have to make sure that the system has the prerequisites from the theorem – and it is also cheating: after all we have just proven that Consis($\mathcal{S}$) can not be deduced and have not proven it for possible independent predicates with the same *meaning*. So maybe it is not absolutely sure that you can't prove consistency inside the system, but one shouldn't spend too much time trying...

5.2.1 Remarks on the chapter *„The incompleteness of elementary number theory“*

In this chapter a lot of proofs were just sketched and in some places the arguments were pretty vague. But a precise way to deal with these things would not have been possible in the amount of time we could dedicate to it. Nevertheless it would have been a pity not to see the beautiful theorems and the interesting connections between algorithms and computability on one side and proof systems and provability on the other.
If this little digression succeeded in making you interested in a lecture on *proof theory*, *logic* or *foundations of mathematics*, this is a success justifying the effort.

67 Exercise *Assume that Z_E is correct. Prove: the set $\{w|w$ is deducible in $Z_E\}$ is not decidable.*

6 Complexity classes

In this chapter we will take an *a bit* more realistic view on things. Until now we only distinguished between *computable* and *not computable* – most of the time without taking into account how long a function that is computable needs to be computed. A function needing $10^{10^{10^{10}}}$ years already for small inputs is theoretically computable, but in practice as impossible to compute as a not computable function.
The machines we discussed so far work in a deterministic way. That means that no matter how often they are started on some input they always have the same output. We will now discuss machines that work in a randomized way – that is: started on the same input they can have different outputs. For practical purposes these machines are **completely useless**! If you have a decision problem, it is for example possible that once they output *yes* or 1 and once they output *no* or 0 – so such a machine doesn't even define a function. **But:** these machines are very important to define and investigate a class of problems that is extremely important in practice: the class of *NP-complete* problems. So these machines are important **tools** for investigating practically relevant problems. As real world machines they would be useless.

108 Definition *A non-deterministic* k-tape Turing machine *is a 5-tuple*

$$M = (Z, A, \delta, z_0, E)$$

with the following properties:
Z, A, z_0, E have the same properties as for deterministic (normal) Turing machines and

- *δ is a relation $\delta \subseteq Z \times A^k \times Z \times A^k \times \{L, R, N\}^k$ that we call the transition relation.*

This definition also implies that each deterministic Turing machine is also a non-deterministic Turing machine – only with the special property that for each beginning $(z, a_1, \dots, a_k, \dots)$ there is at most one element in the relation.

109 Definition *Configurations and successor configurations for non-deterministic Turing machines are defined analogous to Turing machines. We can also write $k \vdash k'$ for two configurations k and k'.* **But** *for non-deterministic Turing machines this does not mean that k' is the only possible successor configuration – it is just one possible successor configuration.*
Also the words computation *and* canonical computation *$c_0, \dots, c_t$ are defined analogous to the case of deterministic Turing machines – only that here you*

may have more than one possible computation resp. canonical computation with the same begin computation.

18 Example *The following non-deterministic* 1*-tape Turing machine started on an empty tape writes a sequence of zeros and ones onto the tape. Each finite* 0,1*-sequence is a possible result of this machine. We also say that the machine* guesses *a* 0,1*-sequence.*
The probability of a certain sequence is not important in this context – and also not defined as we do not have probabilities for the different possible transitions.

- $Z := \{z_0, z_e\}$
- $A := \{0, 1, B, \#\}$
- $E := \{z_e\}$
- $\delta := \{(z_0, B, z_0, 0, L), (z_0, B, z_0, 1, L), (z_0, B, z_e, B, R)\}$

Possible computations started on an empty tape are

- z_0, z_eB
- $z_0, z_0B1, z_0B11, Bz_e11$
- $z_0, z_0B0, z_0B00, z_0B100, z_0B1100, z_0B01100, z_0B101100, Bz_e101100$

110 Definition *In this part we will only discuss decision problems. So we modify the machines a bit. Instead of interpreting outputs* 1 *and* 0 *as* yes *and* no*, we will work with two stop states: an accepting stop state* "accept" *and a rejecting stop state* "reject". *It is obvious that – in both* directions *– a machine of one kind can be transformed to one of the other. So the class of computable functions is not changed by this convention and also the complexity is only modified in a minor way (at most a constant factor for time bounds that are at least the length of the input). The intention is again just to make things easier. We call such a machine an acceptor.*
A finite computation stopping in an accepting state is called an accepting computation and a computation stopping in a rejecting state is called a rejecting computation.

If M is an acceptor with alphabet A then the language accepted by M is the set
$L(M) := \{w \in A_{\_}^* | \text{There is an accepting computation of } M \text{ started on } w\}$

Note that for non-deterministic acceptors accepting **and** *rejecting computations can exist for the same input. According to the definition such inputs are part of the language.*

19 Example *A non-deterministic Turing machine M with*
$L(M) = \{w \in \{0,1\}^* | w$ *is binary representation of a number* $n > 1$ *that is not a prime* $\}$,
could e.g. work as follows:

1.) *if w is not a representation of a binary number n, reject.*

2.) *guess a 0,1-sequence w_1. If w_1 is not a representation of a binary number $n_1 \geq 2$, reject.*

3.) *guess a 0,1-sequence w_2. If w_2 is not a representation of a binary number $n_2 \geq 2$, reject.*

4.) *if $n_1 \cdot n_2 = n$ accept, otherwise reject.*

For each string representing a number $n > 1$ that is not prime, there is an accepting computation – for other strings all computations are rejecting.

111 Definition *Time and tape complexity of non-deterministic Turing machines are defined as follows:*
If M is a non-deterministic acceptor with alphabet A, then we define for $w \in A_\_^*$:

$$t_M(w) := \begin{cases} \mathit{min}\{t | \text{there exists an accepting computation with length } t \text{ of } M \text{ started on } w\} & \text{if } w \in L(M) \\ 0 & \text{else} \end{cases}$$

$$s_M(w) := \begin{cases} \mathit{min}\{s | \text{there exists an accepting computation with tape complexity } s \text{ of } M \text{ started on } w\} & \text{if } w \in L(M) \\ 0 & \text{else} \end{cases}$$

M is called $f()$-time bounded with a function $f : \mathbb{N} \to \mathbb{N}$ if for all $w \in A_\_^*$ we have $t_M(w) \leq f(|w|)$.
M is called $f()$-tape bounded with a function $f : \mathbb{N} \to \mathbb{N}$ if for all $w \in A_\_^*$ we have $s_M(w) \leq f(|w|)$.

Note that the definitions of complexity do not agree for deterministic and non-deterministic Turing machines when applied to Turing machines that are deterministic and therefore also satisfy the definition of a non-deterministic Turing machine, so that both definitions can be applied. If a Turing machine is a deterministic and non-deterministic Turing machine then it is possible that M is time bounded by $f()$ when interpreted as a non-deterministic Turing machine but **not** when interpreted as a deterministic Turing machine, as for non-deterministic Turing machines rejecting computations are not taken into account. In the remainder of the text it will always be clear which definition is applied when talking about deterministic Turing machines. For non-deterministic Turing machines only one definition can be applied.

112 Definition *We will now define sets of languages that we will call* <u>*complexity classes*</u>*: Let a function $f : \mathbb{N} \to \mathbb{N}$ be given.*

- $DTIME(f()) :=$ $\{L|$ *there is an* $O(f())$ *time bounded deterministic Turing machine* M *with* $L(M) = L\}$
- $NTIME(f()) :=$ $\{L|$ *there is an* $O(f())$ *time bounded non-deterministic Turing machine* M *with* $L(M) = L\}$
- $DTAPE(f()) :=$ $\{L|$ *there is an* $O(f())$ *tape bounded deterministic Turing machine* M *with* $L(M) = L\}$
- $NTAPE(f()) :=$ $\{L|$ *there is an* $O(f())$ *tape bounded non-deterministic Turing machine* M *with* $L(M) = L\}$

We also defined tape complexity classes, but due to lack of time we will focus more on the more important time complexity classes. Two classes are especially interesting:

113 Definition
- $P := \bigcup_{k \in \mathbb{N}} DTIME(n^k)$
- $NP := \bigcup_{k \in \mathbb{N}} NTIME(n^k)$

The theorems about simulation of Turing machines that we have proven can be proven completely analogously for non-deterministic Turing machines. They imply that we would get the exactly same classes P and NP if we would require the existence of a 1-tape Turing machine instead of an arbitrary Turing machine.
What we have – due to lack of time – not proven, but what is in fact true, is that we would also get the same classes if we would (with time- and memory-complexity defined in an analogous way) discuss register-machines, which are idealized models for real world computers, but with an infinite memory.

A rough sketch of our approach here is that a problem in P is *relatively easy* and that a problem that is not in P is *comparatively difficult*. This is already one step closer to reality than just distinguishing between computable and not computable, but not totally realistic. Solving a problem that has time complexity $\Theta(n^{1000000})$ is impossible even for small inputs of size, say, $n = 2$.

On the other hand a problem that is not in P, but sometimes needs time $\Theta(2^n)$ **may** be solvable very fast for **almost** all inputs and in practice you might never meet the difficult cases.
But it is definitely a first step closer to reality...

114 Definition *A total function $f : \mathbb{N} \to \mathbb{N}$ is called tape-constructable if there is an $O(f())$ tape bounded Turing machine that started on an input w computes the binary representation of $f(|w|)$.*

115 Remark *Let $f : \mathbb{N} \to \mathbb{N}$ be a function with $f(n) \geq n$.*
Then we have:

- $DTIME(f()) \subseteq NTIME(f())$
- $DTAPE(f()) \subseteq NTAPE(f())$
- $DTIME(f()) \subseteq DTAPE(f())$
- $NTIME(f()) \subseteq NTAPE(f())$
- $P \subseteq NP$
- *if $f()$ is tape constructable, then*
  $NTIME(f()) \subseteq DTAPE(f()) \subseteq DTIME(2^{O(f())})$
  with $DTIME(2^{O(f())})$ the set of all languages in some $DTIME(2^{g()})$ with $g() = c * f() + c'$ for some constants c, c'.

Proof: The first 5 items are easy as each $f()$ time bounded Turing machine is also an $f()$ time bounded non-deterministic Turing machine (with the definition of time complexity for non-deterministic Turing machines) and as in each step at most one new cell is used.

The last item is not as obvious, so we will give a proof here:

Let M be a non-deterministic k-tape Turing machine that is $c \cdot f() + c'$ time bounded.

Then one can define a deterministic $(k + 1)$-tape Turing machine M' that works as follows:

Started on w the machine M' first computes $l = c \cdot f(|w|) + c'$. That is possible with a tape consumption of $O(f())$. Afterward all sequences $p^1, \ldots, p^l$ with each p^i a string of length $(2k + 1)$ with letters from $Z \times A^k \times \{N, R, L\}^k$ are enumerated on tape 1. Each string is interpreted as a protocol of a computation – that is:

- p_1^i is interpreted as state in step i
- for $1 \le j \le k$ the letter p_{1+j}^i is interpreted as the letter that is written onto tape j in step i
- for $1 \le j \le k$ the letter p_{1+k+j}^i is interpreted as the movement of the head on tape j in step i.

For each string, M' tests whether the string can be the protocol of an accepting computation of M started on w. M is simulated on the other tapes until a stop state is reached or the simulation is interrupted as the next step is not a possible next step of M. In each simulation step it is tested whether the next step is possible according to the transition relations of M. If this way the simulation reaches an accepting stop state, M' stops and accepts.

If no simulation leads to an accepting stop state, M' rejects the input. It is obvious that $L(M) = L(M')$ and that M' is $O(f())$ tape bounded.

The last inclusion follows in a way that we have seen several times before – e.g. in Exercise 30 – so we won't give the proof here. Nevertheless writing up a detailed proof would be a very good exercise! The principle is that you look at the number of possible configurations of a given length. If the computation does more steps than there are different computations, the computation of the Turing machine won't stop any more and you can reject.

■

68 Exercise *Assume that we would define the time complexity for a non-deterministic Turing machine as follows:*
Let M be a non-deterministic acceptor with alphabet A and $w \in A_\_^*$. Define

$$t_M(w) := \begin{cases} \mathit{min}\{t \mid \text{there exists an accepting computation with length } t \text{ by } M \text{ started on } w\} & \text{if } w \in L(M) \\ \mathit{min}\{t \mid \text{there exists a rejecting computation with length } t \text{ by } M \text{ started on } w\} & \text{if } w \notin L(M) \text{ and the set is not empty} \\ \text{undefined} & \text{else} \end{cases}$$

Which effect would that have on the complexity classes we defined, if $f(n) > 0 \quad \forall n \in \mathbb{N}$ and if we consider undefined values to be infinite?

69 Exercise *Do not use Definition 111 for the function $t_M(w)$, but*

$$t'_M(w) := \begin{cases} sup\{t \mid \text{there exists an accepting computation with length } t \text{ by } M \text{ started on } w\} & \text{if } w \in L(M) \\ sup\{t \mid \text{there exists a rejecting computation with length } t \text{ by } M \text{ started on } w\} & \text{if } w \notin L(M) \text{ and the set is not empty} \\ \text{undefined} & \text{else} \end{cases}$$

We define the class NP' like NP but use t'_M instead of t_M.
Again we consider undefined values as infinite. Without proof we state also that for a polynomial $p()$ there is a polynomially time bounded Turing machine that started on w can compute the binary representation of $p(|w|)$.

Do we have $NP' = NP$?

70 Exercise *We already noted that for deterministic Turing machines the time complexity measured by the deterministic definition and the non-deterministic definition differ. If you define P_n to be the class analogous to P but using the non-deterministic definition of time complexity. Do we have $P = P_n$?*

116 Remark *Given two polynomials p_1, p_2 and deterministic regular Turing machines M_1, M_2 that are time bounded by p_1 resp. p_2.*
Then the Turing machine $M_{1,2}$ that started on w first works as M_1 started on w and then continues as M_2 on the output of M_1 is time bounded by a polynomial.

Proof: We may assume $p_1(n) \geq n, p_2(n) \geq n$ and that all coefficients in p_1, p_2 are positive – otherwise we can replace them by any positive value and get a new upper bound.

$M_{1,2}$ makes at most $p_1(|w|) + p_2(|f_{M_1}(w)|) \leq p_1(|w|) + p_2(p_1(|w|))$ steps – and that is again a polynomial.

■

This remark directly implies the following corollary:

117 Corollary *Assume that f, g are total computable functions and that polynomials p_f, p_g and Turing machines M_f, M_g exist that compute f resp. g and are p_f resp. p_g time bounded. Then there is a polynomial $p_{f \circ g}$ and a Turing machine $M_{f \circ g}$ that is time bounded by $p_{f \circ g}$ and computes $f \circ g$.*

118 Definition *Let $L, L' \subseteq A_\_^*$ be two languages.*

- *L is called polynomially reducible to L' (notation: $L \leq_p L'$), if $L \leq_r L'$ (by means of f) (see Definition 65) and there is a deterministic polynomially time bounded Turing machine M that computes $f()$.*

- *L and L' are called polynomially equivalent (notation: $L \equiv_p L'$), if $L \leq_p L'$ and $L' \leq_p L$.*

Languages that are polynomially equivalent are – at least on a very abstract level – considered as being *equally difficult*. In practice one would make a difference if the reduction in one direction takes time $O(n)$ and in the other time $O(n^{100000})$...

119 Definition *Let K be a set of languages.*

- *A language L is called K-hard, if for each language $L' \in K$ we have $L' \leq_p L$.*

- *A language L is called K-complete, if L is K-hard and $L \in K$.*

For an arbitrary class K it is not guaranteed that a K-complete language exists. And even if such a language exists, it is in most cases not obvious whether a given language is K-complete.

71 Exercise *Give a complete list of languages in P that are* **not** *P-complete.*

120 Lemma *Let $L \subseteq A_-^*$ be K-hard. Then we have*

- $L \in P \Rightarrow K \subseteq P$

- $K \setminus P \neq \emptyset \Rightarrow L \notin P$

Proof: Assume that $L' \in K$ is polynomially reducible to L by means of f and M_f computes the function f. Let furthermore M_L be a polynomially time bounded acceptor for L.

Then the Turing machine M' that first works as M_f and afterward as M_L accepts exactly the language L'. According to Remark 116 M' is time bounded by a polynomial.

The second item follows directly from the first.

■

121 Corollary *Assume that K is a set of languages, $L, L' \in K$ and L is K-complete.*
Then we have:

- *$L \in P \Leftrightarrow K \subseteq P$*
- *$L \leq_p L' \Leftrightarrow L'$ is K-complete*
- *L' is K-complete $\Leftrightarrow L' \equiv_p L$*

6.1 NP-complete problems

72 Exercise *Is the following definition of the class NP equivalent to Definition 113?*

The class NP is the set of all languages $L \subset A_-^*$ for which a language $L'_L \in P$ and a polynomial q_L exist so that for $w \in A_-^*$ there is a $v \in A_-^*$ with $|v| \leq q_L(|w|)$, so that $w\#v \in L'_L$ if and only if $w \in L$.
Such a v is called a certificate of the fact that $w \in L$.

In the following we will describe a problem that is NP-complete. It was the first problem that was shown to be NP-complete and the only one for which this property was **directly** proven. Afterward NP-completeness was proven for **many** problems – but always making use of Corollary 121, so that all these proofs are based on the proof of the NP-completeness of this first problem.
If you apply Corollary 121, you are dealing with 2 specific problems – and that is of course much easier than for this first problem, where a reduction for an **arbitrary** problem in NP must be described.
To describe this first NP-complete problem, we first need some definitions.

122 Definition *Let V be a set (of variables) and $0, 1 \notin V$.*
Then the set of Boolean expressions is the smallest set with the following properties:

- *$0 \in B$, $1 \in B$, $V \subset B$*
- *if $b_1, b_2 \in B$ then also $(b_1 \vee b_2)$, $(b_1 \wedge b_2)$ and $(\neg b_1)$*

20 Example *$(((x_1 \vee x_2) \wedge 0) \vee (x_3 \wedge (\neg x_2)))$ is a Boolean expression.*

123 Definition *With the operators $\wedge, \vee$ one can construct functions $\{0,1\}^2 \to \{0,1\}$ by defining for $x, y \in \{0,1\}$:*

$$x \wedge y := \begin{cases} 1 & \text{if } x = y = 1 \\ 0 & \text{else} \end{cases}$$

$$x \vee y := \begin{cases} 0 & \text{if } x = y = 0 \\ 1 & \text{else} \end{cases}$$

The operator $\neg$ describes a function $\{0,1\} \to \{0,1\}$ by defining for $x \in \{0,1\}$: $\neg x := 1 - x$.
The functions $\wedge, \vee$ and $\neg$ are called $\underline{\text{conjunction}}$,$\underline{\text{disjunction}}$ and $\underline{\text{negation}}$ (in this order).

Now we will replace variables in Boolean expressions by values from $\{0,1\}$. Applying the operators, this assigns a value also to the whole expression (compare arithmetic expressions).

124 Definition *An $\underline{\text{assignment}}$ is a function $\phi : V \to \{0,1\}$.*

The value of a Boolean expression b with assignment ϕ is defined as follows:

$$\phi(b) := \begin{cases} b & \text{if } b \in \{0,1\} \\ \phi(x) & \text{if } b = x, \qquad x \in V \\ \neg\phi(b_1) & \text{if } b = \neg b_1 \text{ for a Boolean expression } b_1 \\ \phi(b_1) \vee \phi(b_2) & \text{if } b = (b_1 \vee b_2) \text{ for Boolean expressions } b_1, b_2 \\ \phi(b_1) \wedge \phi(b_2) & \text{if } b = (b_1 \wedge b_2) \text{ for Boolean expressions } b_1, b_2 \end{cases}$$

If for an assignment ϕ we have that $\phi(b) = 1$, then we say that ϕ $\underline{\text{satisfies}}$ the expression b. A Boolean expression b is called $\underline{\text{satisfiable}}$ if there exists an assignment that fulfills b.

If the variables in a Boolean expression are a subset of $\{x_1, \ldots, x_k\}$, for some $k \in \mathbb{N}$, then this defines a function $\{0,1\}^k \to \{0,1\}$ in a canonical way.

21 Example *The expression $(x_1 \vee (x_2 \wedge (\neg x_2)))$ is satisfiable – the assignments ϕ with $\phi(x_1) = 1$ satisfy b.*
The expression $(x_1 \wedge (\neg x_1))$ is an easy example of an expression that is not satisfiable.

125 Definition *We define two Boolean expressions b_1, b_2 as equivalent if the functions defined by these expressions are identical, that means $\phi(b_1) = \phi(b_2)$ for all assignments ϕ.*
For this we take the union of the two sets of variables as basis for the space on which the functions are defined.We write shortly $b_1 = b_2$.

126 Lemma

For Boolean expressions a, b, c we have:

- $(a \vee b) = (b \vee a)$ $\qquad$ $(a \wedge b) = (b \wedge a)$
- $(a \vee (b \vee c)) = ((a \vee b) \vee c)$ $\qquad$ $(a \wedge (b \wedge c)) = ((a \wedge b) \wedge c)$
- $(a \vee (a \wedge b)) = a$ $\qquad$ $(a \wedge (a \vee b)) = a$
- $(a \vee (b \wedge c)) = (a \vee b) \wedge (a \vee c)$ $\qquad$ $(a \wedge (b \vee c)) = (a \wedge b) \vee (a \wedge c)$
- $(a \vee 0) = a$ $\qquad$ $(a \wedge 1) = a$
- $(a \vee (\neg a)) = 1$ $\qquad$ $(a \wedge (\neg a)) = 0$

Proof: Just fill in the definition of the operators…

■

127 Definition *If M is a set and $0, 1 \in M$ (in our case $M = \{0, 1\}$) and if $\wedge, \vee : M^2 \to M$ and $\neg : M \to M$ are functions so that Lemma 126 is valid for these functions, then $(M, \vee, \wedge, \neg)$ is called a Boolean algebra.*

22 Example *If $K \neq \emptyset$ is a set, then each system of subsets that is closed under forming the union, the intersection and the complement forms a Boolean Algebra. In this algebra 0 is the empty set, intersections implement $\wedge$, unions implement $\vee$ and the complement implements the operator $\neg$.*

Notation
For $\neg x$ we also write $\bar{x}$.

128 Definition *An expression of the form $(y_1 \vee y_2 \vee \cdots \vee y_s)$ with the property that for all $1 \leq i \leq s$ we have that $y_i \in V \cup \{0, 1\}$ or $y_i = \bar{z}_i$ with $z_i \in V$ is called a clause with s literals.*

A Boolean expression of the form $K_1 \wedge \cdots \wedge K_t$ with clauses $K_1 \ldots K_t$ is called a conjunctive normal form and is also said to be in conjunctive normal form.

73 Exercise *Strictly speaking* a Boolean expression in conjunctive normal form *is not really a Boolean expression. But can you justify that we still write it that way?*

129 Lemma *For each Boolean expression b with n variables there is an equivalent Boolean expression in conjunctive normal form with at most 2^n clauses and each clause has at most n literals.*

Proof: If $\phi(b) = 1$ for all assignments then let $b' = 1$. Otherwise let the variables be $x_1, \ldots, x_n$.

For each assignment ϕ with $\phi(b) = 0$ we build a clause that is 0 for this and only this assignment:

If ϕ is an assignment with $\phi(b) = 0$, then define a clause $K(\phi) := (y_1 \vee \cdots \vee y_n)$ as

$$y_i := \begin{cases} x_i & \text{if } \phi(x_i) = 0 \\ \bar{x}_i & \text{if } \phi(x_i) = 1 \end{cases}$$

This clause is 0 for ϕ and is 1 for all other assignments. Now let b' be the expression in conjunctive normal form that contains exactly the clauses for all the assignments ϕ with $\phi(b) = 0$.

If $\phi(b) = 0$ then we also have $\phi(b') = 0$. If on the other hand $\phi(b) = 1$ then all clauses in b' have value 1 – so $\phi(b') = 1$.

The expression b' in this form is called the complete conjunctive normal form.

■

Now we can define the first language which was proven to be NP-complete and on which all NP-completeness results are based:

130 Definition $V := \{xw | w \in \{0,1\}^*\}$
$A := \{0, 1, (,), x, \vee, \wedge, \neg\}$
$\mathcal{SAT} := \{b \in A^*_- |$ *b codes a satisfiable Boolean expression in conjunctive normal form* $\}$

131 Theorem *(Cook (published 1971), Levin (published 1973))*
$\mathcal{SAT}$ is NP-complete.

Proof: The easy part is to show that $\mathcal{SAT} \in NP$: Started on $w \in A^*_-$ it is first checked whether w is (or to be exact: codes) a Boolean expression in conjunctive normal form. If not, w is rejected. Otherwise the variables are searched and each time one is found it is replaced by a random value from $\{0, 1\}$ (in all places where this variable occurs). When all variables have been replaced by values the expression is evaluated. If the value is 1, w is accepted, otherwise rejected.

This machine accepts exactly the elements of $\mathcal{SAT}$ and it would be relatively easy to describe e.g. a 2-tape Turing machine that runs in time $O(|w|^2)$.

It is **much** more difficult to show that $\mathcal{SAT}$ is NP-hard.

Assume an arbitrary $L \in NP$ to be given. **All** we know of L is that there is a polynomial $p(n)$ and a $p(n)$ time bounded non-deterministic 1-tape Turing machine $M_0 = (Z, A, \delta, z_0, \{z_+, z_-\})$ accepting L with z_+ the accepting stop state and z_- the rejecting stop state.

This information must be sufficient to define our reduction. We must use that for each $w \in L$ an accepting computation with length at most $p(|w|)$ of M_0 started on w exists.

We modify M_0 so that the accepting stop state does not really stop, but goes to an infinite loop without moving and without changing the state. This means just adding the rule (z_+, a, z_+, a, N) for all $a \in A$ to the transition rules. We call the modified machine M.

Now we will construct a Boolean expression in conjunctive normal form that is satisfiable if and only if there is an accepting computation of M_0 started on w – or equivalently: if there is a computation so that M started on w is in z_+ after $p(|w|)$ steps. This Boolean expression in conjunctive normal form must be computable by a deterministic Turing machine started on w that is time bounded by a polynomial in $n = |w|$.

There will be a 1-1 correspondence between satisfying assignments of the Boolean expression and accepting computations with length at most $p(|w|)$.

If we assign numbers to the cells of the tape with number 0 for the cell where the head is on when the computation starts, then M can only visit cells with numbers $-p(n)$ to $p(n)$.

We will now invent variables $c_{i,a,t}$ for $i \in \{-p(n), \ldots, p(n)\}$, $a \in A$ and $t \in \{0, \ldots, p(n)\}$. Later we will translate an accepting computation to a satisfying assignment. To this end a satisfying assignment of the expression will be assigned 1 for variable $c_{i,a,t}$ if cell number i contains letter a in step t for the computation corresponding to that assignment.

The set of all such variables for given i and t is denoted as $C_{i,t}$. We also write $C_t = \bigcup_{i \in \{-p(n),\ldots,p(n)\}} C_{i,t}$.

In order to be able to interpret an assignment as corresponding to a computation, exactly one $c_{i,a,t}$ must be one in each $C_{i,t}$ – as in each time step of a computation there is exactly one letter in each cell i.

We use variables $h_{i,t}$ with $-p(n) \leq i \leq p(n)$, $0 \leq t \leq p(n)$ to code the positions of the heads. The set of all $h_{i,t}$ with a given t is called H_t.

If for an assignment exactly one $h_{i,t}$ is 1 and all the others 0, we can interpret that as *"in step t the head is on position i"*.

In a completely analogous way we use $s_{z,t}$ with $z \in Z$,$0 \leq t \leq p(n)$ to code the state in step t. The set of all $s_{z,t}$ with given t is called S_t. If for a given assignment there is exactly one $s_{z,t}$ in S_t with value 1 (and all others have value 0) we can interpret that as *"in step t the machine is in state z"*.

Now we will describe how to construct a Boolean expression in conjunctive normal form that can guarantee that we can interpret a satisfying assignment of the variables just defined as an accepting computation:

First we will guarantee that a satisfying assignment fulfills the uniqueness requirements already described.

If $x_1, \ldots, x_n$ is an arbitrary set of variables, then

$$u(x_1, \ldots, x_n) := (x_1 \vee x_2 \vee \cdots \vee x_n) \wedge \bigwedge_{i \neq j} (\bar{x}_i \vee \bar{x}_j)$$

is an expression that is satisfied by an assignment if and only if exactly one of the variables is assigned 1.

The following expression is satisfied by an assignment ϕ if and only if we can interpret the values of the variables for step t as describing a configuration $k_t(\phi)$ at step t of the Turing machine.

$$\mathrm{conf}_t := \mathrm{conf}(C_t \cup H_t \cup S_t) := u(S_t) \wedge u(H_t) \wedge \bigwedge_{-p(n) \leq i \leq p(n)} u(C_{i,t})$$

The part $u(H_t)$ has quadratic length (in $p(n)$), the part $\bigwedge_{-p(n) \leq i \leq p(n)} u(C_{i,t})$ linear length and the part $u(S_t)$ constant length.

So altogether the expression conf_t has a length of $O(p^2(n))$. conf_t is satisfied if and only if there is precisely one $h_{i,t}$, precisely one $s_{z,t}$ and for each i precisely one $c_{i,a,t}$ that is assigned a 1.

Now we can interpret a satisfying assignment as a series of configurations $c_1(\phi) \ldots c_{p(n)}(\phi)$ – but they don't have to be successor configurations of each other – so a satisfying assignment need not represent a computation.

First we will construct an expression guaranteeing that for two successive configurations for the sets C_t, C_{t+1} at most the content of the cell differs where the head was at time t:

The expression $(x \equiv y) := ((\bar{x} \vee y) \wedge (x \vee \bar{y}))$ is 1 for an assignment ϕ if and only if $\phi(x) = \phi(y)$.

The expression $(h_{i,t} \vee (c_{i,a,t} \equiv c_{i,a,t+1}))$ is not in conjunctive normal form, but we can replace it by an equivalent expression $I(i,a,t)$. As the length of $(h_{i,t} \vee (c_{i,a,t} \equiv c_{i,a,t+1}))$ is constant – so independent of the input – also the length of $I(i,a,t)$ is constant.

$$D_t := \bigwedge_{a \in A} (\bigwedge_{-p(n) \leq i \leq p(n)} I(i,a,t))$$

has length $O(p(n))$ and is 1 for an assignment if and only if for each position i the contents at time steps t and $(t+1)$ are identical or the head is on cell i in time step t.

In $\mathrm{conf}_t \wedge \mathrm{conf}_{t+1} \wedge D_t$ it is furthermore guaranteed that in both time steps a satisfying assignment can be interpreted as a unique letter for each cell and a unique cell for the head.

So far we did not speak about the transition relation. Now we will define an expression that guarantees that for a satisfying assignment we can interpret configurations $c_t(\phi)$ and $c_{t+1}(\phi)$ as successive steps of our machine M.

For $-p(n) \leq i \leq p(n)$,$a \in A$,$z \in Z$ and $0 \leq t \leq p(n)$ define

$\delta^1_{i,a,z,t} := \bar{c}_{i,a,t} \vee \bar{h}_{i,t} \vee \bar{s}_{z,t}$

For a given assignment this expression is 1 if at the time step t the letter a is not in cell i, the head is not on position i or the state is not z.

Let also (**first** not in conjunctive normal form):

$$\begin{aligned}\delta^2_{i,a,z,t} := \quad & \bigvee\nolimits_{(z,a,q,b,R)\in\delta}(c_{i,b,t+1} \wedge h_{i+1,t+1} \wedge s_{q,t+1}) \vee \\ & \bigvee\nolimits_{(z,a,q,b,L)\in\delta}(c_{i,b,t+1} \wedge h_{i-1,t+1} \wedge s_{q,t+1}) \vee \\ & \bigvee\nolimits_{(z,a,q,b,N)\in\delta}(c_{i,b,t+1} \wedge h_{i,t+1} \wedge s_{q,t+1})\end{aligned}$$

For $i = p$ we remove the part
$\bigvee_{(z,a,q,b,R)\in\delta}(c_{i,b,t+1} \wedge h_{i+1,t+1} \wedge s_{q,t+1}) \vee$
as $h_{p+1,t+1}$ does not exist and analogously for $i = -p$,

$\bigvee_{(z,a,q,b,L)\in\delta}(c_{i,b,t+1} \wedge h_{i-1,t+1} \wedge s_{q,t+1})\vee$
and $h_{-p-1,t+1}$.

Also the lengths of $\delta^1_{i,a,z,t}$ and $\delta^2_{i,a,z,t}$ are independent of the input – and therefore constant. So there is an expression $\delta_{i,a,z,t}$ in conjunctive normal form that is equivalent to $\delta^1_{i,a,z,t} \vee \delta^2_{i,a,z,t}$ and also has constant length.

This expression is satisfiable for a 4-tuple i, a, z, t if and only if in order to decide whether the assignement can represent a computation the transition relation does not have to be tested ($\delta^1 = 1$) or the transition that took place is covered by one of the transition relations of M.

$$\delta_t := \bigwedge_{z\in Z}(\bigwedge_{a\in A}(\bigwedge_{-p(n)\leq i\leq p(n)} \delta_{i,a,z,t}))$$

has therefore a length of $O(p(n))$.

With these expressions we have

If ϕ is an assignment satisfying

$\text{conf}_t \wedge \text{conf}_{t+1} \wedge D_t \wedge \delta_t$

then $c_t(\phi) \vdash c_{t+1}(\phi)$.

The expression

$$A(w) := \bigwedge_{-p(n)\leq i<0} c_{i,B,0} \wedge \bigwedge_{0\leq i<n} c_{i,w_{i+1},0} \wedge \bigwedge_{n\leq i\leq p(n)} c_{i,B,0} \wedge h_{0,0} \wedge s_{z_0,0}$$

is satisfied by an assignment ϕ, if and only if $c_0(\phi) = B^{p(n)}z_0wB^{p(n)-n+1}$ – that is $c_0(\phi)$ corresponds to a start configuration of M started on w.

Finally we define

$$F(w) := A(w) \wedge (\bigwedge_{0\leq t\leq p(n)} \text{conf}_t) \wedge (\bigwedge_{0\leq t<p(n)} (D_t \wedge \delta_t)) \wedge s_{z_+,p(n)}$$

For an assignment ϕ that satisfies $F()$ – so $\phi(F(w)) = 1$ – we have $c_0(\phi), \ldots, c_{p(n)}(\phi)$ is a possible computation of M so that the state of M in time step $p(n)$ is z_+. So there is a $j \leq p(n)$ so that $c_0(\phi), \ldots, c_j(\phi)$ is an accepting computation of M_0 started on w.

The other direction – that if an accepting computation of M_0 started on w of length at most $p(n)$ exists, then $F(w)$ is satisfiable, is easy. Using an accepting computation $c_1, \ldots, c_{p(n)}$ van M_0 we can construct ϕ:

$$\phi(c_{i,a,t}) := \begin{cases} 1 & \text{if in cell } i \text{ of } c_t \text{ we have letter } a \\ 0 & \text{else} \end{cases}$$

$$\phi(h_{i,t}) := \begin{cases} 1 & \text{if in } c_t \text{ the head is on cell } i \\ 0 & \text{else} \end{cases}$$

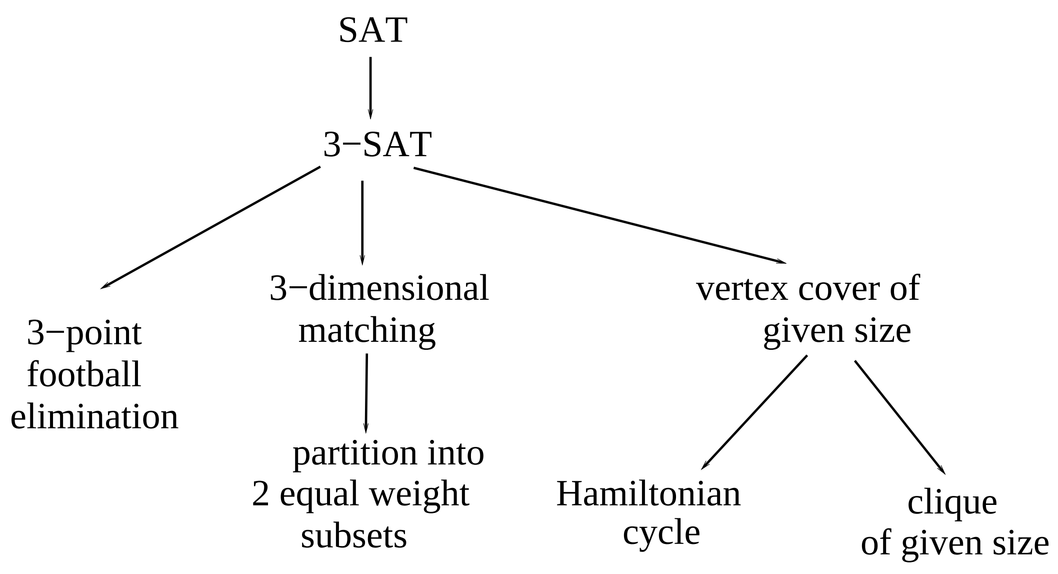
$$\phi(s_{z,t}) := \begin{cases} 1 & \text{if the state in } c_t \text{ is } z \\ 0 & \text{else} \end{cases}$$

The length of $F(w)$ is $O(p^3(n))$. The fact that $F(w)$ can in fact be constructed by a polynomially time bounded Turing machine is *relatively clear* as we have now quite some experience with what a Turing machine can do and can not do. But really writing up the details would of course be "a bit" lengthy.

In the proof the variables got names that help to understand the proof a bit better. In the translation the real names would be from the set $\{xw | w \in \{0,1\}^*\}$. This would make $F(w)$ still a bit longer as so far we counted the variables as taking one letter. But the increase in length would be a factor of at most $c \cdot \log(p(n))$.

■

In this proof the machine that solved the problem was much more important than the problem itself. But that is of course not surprising: after all, all we knew about the problem was that there existed a machine solving it...
If an **explicit** problem is given, things are very different: If L is an NP-complete language and $L' \in NP$ another explicitly given problem (that is: language), then we can try to prove that also L' is NP-complete by reducing L to L'. For this we use the languages and not the machines solving the problems (that is: accepting the languages).
The following figure shows for some problems how the chain of reductions proving NP-completeness works:


SAT
3−SAT
3−point football elimination
3−dimensional matching
vertex cover of given size
partition into 2 equal weight subsets
Hamiltonian cycle
clique of given size

74 Exercise $V := \{xw | w \in \{0,1\}^*\}$
$A := \{0, 1, (,), x, \vee, \wedge, \neg\}$
$\mathcal{SAT}_0 := \{b \in A_\_^* | b$ *codes a satisfiable Boolean expression*$\}$

Here we do not require the expression to be in conjunctive normal form.
Prove that $\mathcal{SAT}_0$ *is NP-complete.*
Why do you think that $\mathcal{SAT}$ *is the origin of all the reducibility results – why didn't people start with this language* $\mathcal{SAT}_0$*?*

132 Theorem
$V := \{xw | w \in \{0,1\}^*\}$, $A := \{0, 1, (,), x, \vee, \wedge, \neg\}$

3-$\mathcal{SAT} := \{b \in A_\_^* |$ *b codes a satisfiable Boolean expression in conjunctive normal form where each clause contains exactly 3 literals*$\}$

is NP*-complete.*

Proof: The fact that 3-$\mathcal{SAT}$ as a subset of $\mathcal{SAT}$ with an additional structure that can easily be tested can be reduced to $\mathcal{SAT}$ is obvious. Unfortunately that is not the direction we need...

The easy part is again (like for most NP-complete problems) to show that 3-$\mathcal{SAT} \in NP$. This can be done completely analogously to $\mathcal{SAT}$ except for the additional test whether each clause contains eactly 3 literals.

In order to show that 3-$\mathcal{SAT}$ is NP-hard, we will reduce $\mathcal{SAT}$ to 3-$\mathcal{SAT}$.

If the input is not coding an expression in conjunctive normal form, we leave it unchanged – the input will neither be an element of $\mathcal{SAT}$ nor of 3-$\mathcal{SAT}$.

Otherwise we will construct a new Boolean expression b' for each Boolean expression b in conjunctive normal form , so that b' is also in conjunctive normal form, has 3 literals in each clause and is satisfiable if and only if b is. It will be possible to do the construction by a deterministic Turing machine that is time bounded by a polynomial in the size of b.

We will replace each clause $(x_1 \vee x_2 \vee \cdots \vee x_n)$ of b by an expression in conjunctive normal form where each clause has 3 literals. For this we need some additional variables y_i that are different for pairwise different clauses of b – but we will omit the additional index j making them unique for each clause for the sake of more easy understanding.

We look at the different cases of numbers of literals in the clause $(x_1 \vee x_2 \vee \cdots \vee x_n)$:

$n = 1$: We could just add two "$\vee 0$" to the clause, but we can also do it without constants.
For this we need two extra variables: y_1, y_2.
Replace (x) by $(x \vee y_1 \vee y_2) \wedge (x \vee \bar{y}_1 \vee y_2) \wedge (x \vee y_1 \vee \bar{y}_2) \wedge (x \vee \bar{y}_1 \vee \bar{y}_2)$

$n = 2$: Either add "$\vee 0$" to the clause or use an extra variable: y_1.
In the second case replace $(x_1 \vee x_2)$ by $(x_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \bar{y}_1)$

$n = 3$: These clauses can remain as they are.

$n \geq 4$: We use extra variables: $y_1, \ldots, y_{n-3}$.
Replace $(x_1 \vee \cdots \vee x_n)$ by $(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{n-4} \vee x_{n-2} \vee y_{n-3}) \wedge (\bar{y}_{n-3} \vee x_{n-1} \vee x_n)$

For $n \in \{1, 2, 3\}$ it is immediately clear that a satisfying assignment for the new clause can be translated to one for the old with the same value of the x_i in both expressions. The other direction is equally clear.

But the case $n \geq 4$ needs some explanation:

If $(x_1 \vee \cdots \vee x_n)$ is satisfied then there is a literal x_j with value 1.

If $j \in \{1, 2\}$ one can choose all y_i as 0 and if $j \in \{n-1, n\}$ one can choose all y_i as 1 to have a satisfying assignment for the clauses replacing $(x_1 \vee \cdots \vee x_n)$ with the same values for the x_i.

So assume $3 \leq j \leq n-2$. The variables $y_{j-1}, \bar{y}_{j-2}$ are in the same clause as x_j.

Assign 0 to all y_k with $k \geq j-1$ and assign 1 to all y_k with $k \leq j-2$.

Then in each clause – except $(\bar{y}_{j-2} \vee x_j \vee y_{j-1})$ that is already satisfied – there is a literal corresponding to a y_i with value 1. So the assignment to the whole new expression has value 1.

Now the other direction:

If a satisfying assignment of $(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{n-4} \vee x_{n-2} \vee y_{n-3}) \wedge (\bar{y}_{n-3} \vee x_{n-1} \vee x_n)$ is given, we have to show that one of the literals x_i has value 1, so that also $(x_1 \vee \cdots \vee x_n)$ is satisfied. Assume that all have value 0. Then y_1 has value 1, in order to satisfy the first clause and y_{n-3} has value 0 to satisfy the last clause. So there is a k so that y_k is assigned 1 and y_{k+1} is assigned 0. But then the value of the clause $(\bar{y}_k \vee x_{k+2} \vee y_{k+1})$ and therefore of the whole expression is 0 – a contradiction. So one of the literals x_i $1 \leq i \leq n$ has value 1 and $(x_1 \vee \cdots \vee x_n)$ is satisfied.

You can easily convince yourself that the resulting formula has polynomial length in the length of the input formula and that the translation can be done in polynomial time.

■

75 Exercise

2-$\mathcal{SAT} := \{b \in A_{\_}^{*} \mid$ *b codes a satisfiable Boolean expression in conjunctive normal form where each clause contains exactly 2 literals*$\}$

Give a (deterministic) polynomial algorithm to decide whether a given input belongs to 2-$\mathcal{SAT}$.

Or with other words: prove that 2-$\mathcal{SAT} \in P$

Finally we will prove NP-completeness of one more problem. It is not one of the classical NP-complete problems like the Hamiltonian Cycle problem, the Travelling Salesman Problem, etc. As they are typical graph problems, proving NP-completeness of these problems will be (a small) part of the lecture *Algorithmic Graph Theory*. The problem here has the nice property that – at first sight – it has nothing to do with $\mathcal{SAT}$, 3-$\mathcal{SAT}$ or mathematics. But when you have seen the reduction, you suddenly do see similarities, and this also makes it a nice problem: there is not just technique, but really something to understand...

We will not explicitly describe how to code the problem (the words of the language) as 0-1-sequences. This can be done in lots of different ways and isn't difficult. Each of you could immediately come up with a way how that can be done.

Description of the problem.

133 Definition *Let T be a finite set (of football teams).*
Let $G \subset T \times T \times \mathbb{N}$ be a finite set (of football matches between these teams) with $t_1 \neq t_2$ for all $(t_1, t_2, i) \in G$.
The last entry of the triple is just to be able to generalize it further. In a normal football competition you only have two matches between two teams t_1, t_2, so they can be described as $(t_1, t_2), (t_2, t_1)$. In other competitions you can have more matches between two teams and then you can use the last entry to distinguish between them.

Let $P \subset \mathbb{N} \times \mathbb{N}$ be a finite set (of possible distributions of points for a match). Some time ago we had $P = \{(2,0), (1,1), (0,2)\}$ (2-point rule). A winner got 2 points, a loser got no points and in case of a tie, both teams got 1 point. Now there is the 3-point rule $P = \{(3,0), (1,1), (0,3)\}$: a winner gets 3 points, a loser gets no point and in case of a tie, both teams get 1 point.

A season is a function $S : G \to P$.
In reality this function is determined by the outcome of the matches – who wins or whether it was a tie.

For $t \in T$, $g = (t_1, t_2, i) \in G$ and
$S(t_1, t_2, i) = (x, y)$ let

$$p_S(t, g) = \begin{cases} x & \text{if } t = t_1 \\ y & \text{if } t = t_2 \\ 0 & \text{else} \end{cases}$$

$p_S(t, g)$ is called the number of points won by team t in match g.

If $G' \subseteq G$ then the function $S' : G' \to P$ is called an intermediate table and a mapping $S'' : G \setminus G' \to P$ is called a scenario for the remaining games.

23 Example *In the* Jupiler Pro League *in Belgium we have $T = \{t_1, \ldots, t_{16}\}$ (with the names of the teams still to be filled in instead of the t_i) and $G = T \times T \times \{1\} \setminus \{(t, t, 1) | t \in T\}$.*

134 Definition *The sport elimination problem is defined as follows:*
Given a subset $G' \subseteq G$ of the set of games, an intermediate table $S' : G' \to P$ and a team $t \in T$.

Is there a season $S : G \to P$
with $S|_{G'} = S'$ so that

$$\sum_{g \in G} p_S(t,g) \geq \sum_{g \in G} p_S(t',g) \qquad \forall t' \in T$$

In this case we say that team t is not yet elimated for the championship.

A bit more formal one could write:
Assume P to be given

$L := \{w \in \{0,1,\#\}^* |$ *$w = w_1\#w_2, w_1, w_2 \in \{0,1\}^*$ with w_1 coding T, G and an intermediate table, and w_2 coding a team $t \in T$ so that t is not yet eliminated for the championship.*$\}$

Of course we have to fix a way to code teams, matches and intermediate tables to make this description really formal.

Informally you could say that it means that the remaining matches can *in principle* finish with results in a way that at the end of the season no team has more points than t.

135 Theorem *(Schwarz 1966)*
The sport elimination problem for the 2-point rule is in P.

Proof: Schwarz reduces the problem to a maximum flow problem in directed graphs. We won't describe that here and will refer interested students again to *Algorithmic Graph Theory*.

■

136 Theorem *(Bernholt,Gülich,Hofmeister,Schmitt 1999)*

- *The sport elimination problem for the 3-point rule is NP-complete.*
- *Even if we restrict it to the subset of intermediate tables where each team has at most 3 matches left to be played, the problem is NP-complete.*

Proof: We prove only the second – stronger – statement. That implies the first one immediately as the second problem can be reduced to the first one:

First the structural requirements for the code in the more restricted case are tested and if the input does not fulfill these (e.g. too many matches left, or not coding teams, or...), an invalid code for the more general problem (not coding teams, matches, etc.) is written as output. Otherwise the identical mapping is used as the reduction.

Again it is easy to see that the problem is in NP: guess a scenario for the remaining games, evaluate that season and accept if at the end no team has more points than t – otherwise reject.

We will first make a small change in order to make some arguments more easy (without really changing the problem). This is in principle a trivial (and definitely polynomial) reduction to another problem that works in both ways. The inverse of what we will describe would be needed as a reduction to show that the original problem is NP-complete.

For each match that has not yet been played, each participating team gets one point. For the remaining matches in which a team t participates, the point distribution is then as follows:

tie: 0 points for team t

t **wins:** 2 points for t

t **loses:** -1 points for t

After all matches have been played, the teams have exactly as many points as without this modification.

We will decide whether team number n is eliminated. If one has to decide on another team, the teams would just have to be relabeled.

We may assume that team t_n has already played all matches. Otherwise we can just assume that it wins all the remaining matches as team t_n is eliminated if and only if it is eliminated after having won all remaining games. This is true for this case, but you should think about which property of the point distribution is necessary for this to be true (and what *wins* means for a general point distribution...?).

First we will describe the problem by a multigraph with labeled vertices. The vertices are the teams and the labels of a vertex representing team t_k are the points of t_k minus those of t_n – so the negative of the number of points that t_k may still win without having more points than t_n. A remaining match between teams t_i and t_j is represented by an edge between t_i and t_j.

Team t_n is not yet eliminated if by removing edges and updating labels according to the rules for point distributions, at the end one can reach a situation where all vertices have labels $x \leq 0$.

This graph is nothing new – it is just a way to represent the problem – but that is in principle true for every translation.

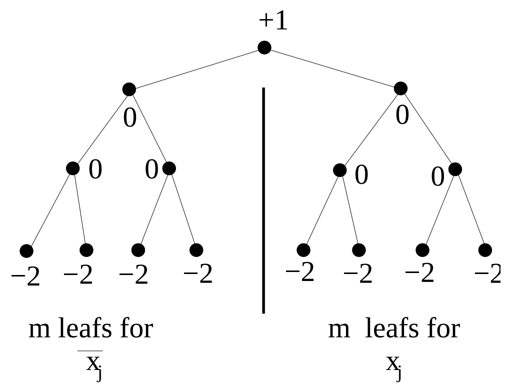
Now we will reduce 3-$\mathcal{SAT}$ to this problem:

Assume that an input is given which has to be tested to decide whether it belongs to 3-$\mathcal{SAT}$. Assume that the trivial cases that don't fulfill the structural requirements have already been decided, so that we have a Boolean expression in conjunctive normal form with 3 literals in each clause. Assume also that the same literal never occurs twice in the same clause. Otherwise we replace it just like in the proof that 3-$\mathcal{SAT}$ is NP-complete.

Assume also that the number of clauses is m, with m a power of 2. If this would not be the case we could just add clauses that are always fulfilled until it is a power of two. The result would be at most 2 times more clauses.

We will now build a labeled graph with a size that is a polynomial in the size of the Boolean expression and that can be constructed (and coded) by a polynomially time bounded Turing machine. This graph will represent a situation where team t_n will not be eliminated if and only if the Boolean expression is satisfiable.

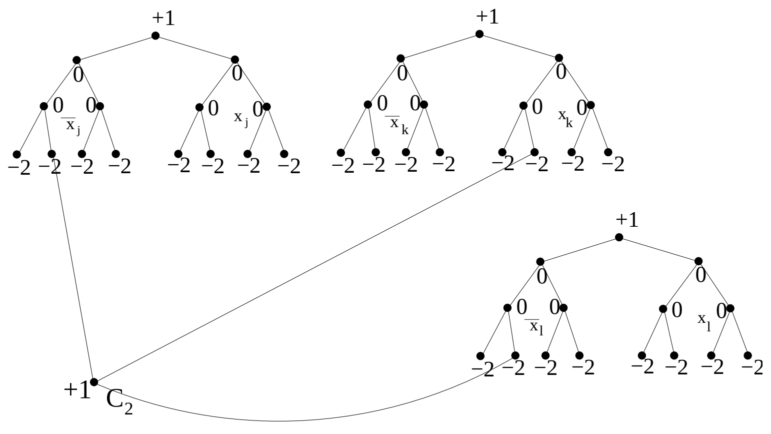
For each variable x_j we build a binary tree as follows:



The remarks “for x_j” resp. “for $\bar{x}_j$” are just to help understanding what we will do now.

Then for each clause C_i we use a vertex C_i with label $+1$. It is connected to the variable trees of the variables that are contained in the clause as follows: If x_j is in the clause, we connect C_i with the i-th leaf of the $\bar{x}_j$ part of the x_j-tree. If $\bar{x}_j$ is in the clause, we connect C_i with the i-th leaf of the x_j part of the x_j-tree.

Example: $C_2 = (x_j \vee \bar{x}_k \vee x_l)$



We will first show that if the Boolean expression is satisfiable, then there is also a scenario for the remaining games so that no team gets more points than t_n.

If ϕ is an assignment that satisfies the expression, fix the results of the remaining games as follows:

If $\phi(x_i) = 1$ define all matches in the x_i-subtree as won by the child and all matches in the $\bar{x}_i$-subtree as ties.

If $\phi(x_i) = 0$ define all matches in the $\bar{x}_i$-subtree as won by the child and all matches in the x_i-subtree as ties.

What remains are the matches in which C_i participates:

As at least one literal in C_j has value 1, C_j is connected to at least one leaf that has label -2 after these modifications (in that subtree all matches were ties). Define this match as won by the leaf and the other matches as ties.

This way no vertex has a label larger than 0 afterward.

Now the other direction: assume that a scenario is given so that n is not yet eliminated.

In such a scenario the root of the x_j-tree loses at least one match. Define $\phi(x_j) = 0$ if the root loses against the child in the $\bar{x}_j$-subtree and $\phi(x_j) = 1$ otherwise.

If in a subtree one of the children wins, then for all matches on the way to the leafs also the children must win – otherwise one of the internal vertices would have a positive label in the end.

In such a scenario, each clause C_i must lose a match too. So the vertex has a neighbor that is a leaf and sits in a subtree where **not** all matches are won by the children (otherwise the vertex would have positive label in the end). So the clause C_i contains a literal x_j with $\phi(x_j) = 1$ or a literal $\bar{x}_j$ and $\phi(x_j) = 0$. In each case the value of each clause – and therefore also of the whole Boolean expression – is 1 and the Boolean expression is satisfied.

We represented the elimination problem as a graph, but we did not discuss whether each labeled graph can represent an intermediate table. For the graphs used in the proof this can in fact be shown. One can give matches so that the table after this set of matches is exactly the graph in the proof. But that is quite technical and not really beautiful. You need some extra teams that afterward again vanish (as they have already played all matches, etc.). We have seen the beautiful idea of the proof and have understood the strong connection between 3-$\mathcal{SAT}$ and 3-point football elimination. People who want to see the details of this last part are referred to the paper by Bernholt, Gülich, Hofmeister and Schmitt.

■

76 Exercise *Assume that the situation is like in the second part of Theorem 136 – only that each team has at most 2 matches left to be played instead of 3. Give the exact description of this language and show that unless $P = NP$ this language is not NP-complete.*

77 Exercise *It is clear what – analogous to a conjunctive normal form – a disjunctive normal form of a Boolean expression is. Let $\mathcal{SAT}_D := \{b \in A_\_^* | b$ codes a satisfiable Boolean expression in disjunctive normal form$\}$. Assume $P \neq NP$ and prove or disprove that $\mathcal{SAT}_D$ is NP-complete.*

One very easy exercise at the end:

78 Exercise $L := \{w \in \{0,1\}^* | w$ *is a binary representation of an even number* $\}$ *Show: if L is not NP-complete, then $P \neq NP$.*

OK – now we are at the end of the course. We touched a lot of topics in computability theory and complexity theory, but could not go very deep in these topics. For all these topics **a lot** of other results exist and some are a field of their own. For computer scientists maybe the theory of NP-completeness is the most interesting one, also as many problems that are important in real life are NP-complete. As you know the question "$P = NP$?" is one of the Millenium Problems. Solving it you would immediately become famous – and earn 1.000.000 Dollars. But if you do it for the money: better learn to play football, tennis, etc. The chance to become famous and earn millions there is **much** larger. But as a scientist, you simply want to know…

Index

A^*, 35
A^+, 35
A_-, 35
$A_,$ 87
K-complete, 115
K-hard, 115
M_{+1}, 38
$M_{1,n}$, 38
Z_e, 88
$\mathcal{SAT}$, 119
$\cdot$, 12
$\equiv_p$, 115
$\leq_p$, 115
$\leq_r$, 76
μ-operator, 30
$\mu_b f$, 14
μ-recursive, 30
$\vdash$, 36
f_M, 57
f_M^r, 57
$\mathcal{R}$, 30
$+$, 11
2-point rule, 128
3-point rule, 128

computer
    normal, 19
time complexity, 40

A. Church, 68
accepting computation, 109
acceptor, 109
Ackermann function, 23
algebra
    Boolean, 118
alphabet, 35
approximation
    worst case, 41
arithmetic statement, 89
assignment, 90, 117
assignment
    value, 90
axiom, 87
axioms
    elementary number theory, 92

beaver
    busy, 68
bonded, 89
Boolean algebra, 118
Boolean expression, 116
    equivalent, 117
bounded
    memory, 41
    tape, 41
    tape, non-deterministic, 110
    time, 40
    time, non-deterministic, 110
bounded μ-operator, 14
busy beaver, 68

canonical computation, 37
case distinction, 14
certificate, 116
characteristic function, 13
Chrurch-Turing, 68
Church, 68
Church's thesis, 68
Church-Turing thesis, 68
clause, 118
complete, 97, 115
complete conjunctive normal form, 119
complexity
    memory, 40
    tape, 40
    tape, non-deterministic, 110

time, 40
time, non-deterministic, 110
complexity classes, 111
computable
Turing, 57
computation, 37
computation
accepting, 109
canonical, 37
length, 40
memory complexity, 40
rejecting, 109
result, 35
tape complexity, 40
time, 40
time complexity, 40
Turing machine, 35
configuration, 36
non-deterministic, 108
successor, 36
configuration
end configuration, 36
transition, 46
conjunction, 117
conjunctive normal form, 118
conjunctive normal form
complete, 119
consistent, 97
content, 34
content
regular, 39
correct, 97
CPU, 34
CPU
state, 34

David Hilbert, 71
decidable, 72
decision problem, 71
deducible, 87
deduction
elementary number theory, 94
deduction rules, 87
definition
truth, 97
derivation, 11
diagonal argument, 20
difference, 13
diophantic equation, 71
disjunction, 117

elementary number theory
rules of deduction, 94
end configuration, 36
enumerable
recursively, 80
enumerate, 80
equation
diophantic, 71
equivalent
polynomially, 115
expression
satisfiable, 117
expression
Boolean, 116

false, 91
football
season, 128
free, 89
free for, 91
function
constant, 9
derivation, 11
difference, 13
minus, 13
partial, 9
plus, 11
primitive recursive, 9
projection, 9
successor, 9
total, 9

function
  characteristic function, 13
  multiplication, 12
  partial, 30
  power, 13
  predecessor, 12
  signum , 13
  transition, 35
  Ackermann , 23
funktion
  μ-recursive, 30

general halting problem, 76
Gödel number, 73
Gödelisation, 72, 73

Halting problem, 69
halting problem, 72, 75
halting problem
  empty tape, 77
  general, 76
  special, 75
halting problem on empty tape, 77
halting state, 34, 35
hard
  language, 115
Hilbert, 71
  10th problem, 71
Hilbert's 10th problem, 71

intermediate table, 128

Kleene, 67

language, 109
language
  accepted by non-deterministic Turing machine , 109
length, 40
literal, 118

M3, 39
memory bounded, 41
memory complexity, 40
minus, 13
multiplication, 12

negation, 117
normal computer, 19
normal form
  Kleene, 67
number
  Gödel, 73

operator
  μ, 30
  μ_b, 14
  bounded, 14

partial, 9
partial function, 30
plus, 11
polynomially equivalent, 115
polynomially reducible, 115
predecessor function, 12
predicate
  arithmetic, 89
primitive recursion, 10
primitive recursive, 9
  set, 13
problem
  decision, 71
  halting, 72, 75
  halting, special, 75
  sport elimination, 128
projection, 9
proof, 87
proof system, 87
proof system
  complete, 97
  correct, 97
  to contain, 102
  consistent, 97
protocol, 112

Rado, 69
recursion
  primitive, 10
  simultaneous, 18
recursively enumerable, 80
reducible, 76
reducible
  polynomially, 115
reduction, 76
regular, 41, 46
  content, 39
regular content, 39
rejecting computation, 109
relation
  transition, 108
result, 35
Rice, 78
Rice
  theorem of, 78
rules of deduction, 87
rules of deduction
  elementary number theory, 94

satisfiable, 117
satisfy, 117
scenario, 128
season, 128
set
  set of states, 35
set
  characteristic function, 13
  decidable, 72
  primitive recursive, 13
sign function, 13
signum, 13
simulation, 47
simultaneous recursion, 18
special halting problem, 75
sport elimination problem, 128
start state, 34, 35
state, 34, 35
  halting, 34, 35
  start, 34, 35
statement
  arithmetic, 89
  false, 90
  true, 90
stop state
  accepting, 109
  rejecting, 109
substitution, 9
successor configuration, 36
  non-deterministic, 108
successor function, 9

table
  intermediate, 128
tape, 34
  content, 34
  track, 48
tape bounded, 41
tape bounded
  non-deterministic, 110
tape complexity, 40
tape complexity
  non-deterministic, 110
tape-constructable, 112
term, 89
term
  free for, 91
  value, 90
theorem of Rice, 78
time, 40
time bounded, 40
time bounded
  non-deterministic, 110
time complexity
  non-deterministic, 110
total, 9
track, 48
transition configuration, 46
transition function, 35

transition relation, 108
true, 91
truth definition, 97
Turing, 33
Turing computable
    $f : \mathbb{N}^r \to \mathbb{N}$, 57
    $f : A_\_^* \to A_\_^*$, 57
Turing machine, 33
Turing machine
    k-tape, 35
    alphabet, 35
    computation, 35, 37
    configuration, 36
    Definition, 35
    Gödel number, 73
    Gödelisation, 72, 73
    informal, 33
    regular, 41, 46
    non-deterministic, 108
    simulation, 47
    universal, 81
    while, 46

universal Turing machine , 81

value, 90
variable, 89
variable
    bonded, 89
    free, 89

word, 34
worst case approximation, 41