Theoretical Computer Science

An Introduction

Markus Bläser
Universität des Saarlandes

Draft—January 26, 2022 and forever

Contents

Part I

Finite Automata

A Prelude: Mathematical foundations

A.1 Relations

Let A and B be two sets. A *(binary) relation* on A and B is a subset $R \subseteq A \times B$. If $(a, b) \in R$, then we will say that a and b stand in relation R. Instead of $(a, b) \in R$, we will sometimes write aRb. While this looks weird when the relation is called R, let us have a look a the following example.

Example A.1 $R_1 = \{(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)\} \subseteq \{1,2,3\} \times \{1,2,3\}$ *is a relation on* $\{1,2,3\}$.

$(a, b) \in R_1$ means that a is less than or equal to b. In this case, $a \leq b$ looks much better than $(a, b) \in \leq$.

Consider a relation R on a set A and itself, i.e., $R \subseteq A \times A$. R is called *reflexive* if for all $a \in A$, $(a, a) \in R$. R_1 is reflexive, since $(1,1),(2,2),(3,3) \in R_1$.

R is called *symmetric* if $(a, b) \in R$ implies $(b, a) \in R$, too, for all $(a, b) \in A \times A$. The relation R_1 is not symmetric, since for instance $(1, 2) \in R_1$ but $(2, 1) \notin R_1$.

R is called *antisymmetric* if $(a, b) \in R$ and $a \neq b$ implies $(b, a) \notin R$. R_1 is antisymmetric, since we do not consider tuples of the form (a, a). There are relations that are neither symmetric nor antisymmetric. $R_2 = \{(1,2),(1,3),(3,1)\}$ is such an example. It is not symmetric, since $(1, 2) \in R_2$ but not $(2, 1) \in R_2$. It is not antisymmetric, because $(1, 3)$ and $(3, 1)$ are both in R_2.

A relation R is called *transitive* if for all $a, b, c \in A$, $(a, b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$. R_1 is transitive. The only pairs that we have to check are $(1, 2)$ and $(2, 3)$, since these are the only with three different elements and one element in common. But $(1, 3)$ is in R_1, too. The smallest transitive relation $T \subseteq A \times A$ such that $R \subseteq T$ is called the *transitive closure* of R.

A relation that is reflexive, antisymmetric, and transitive is called *partial order*. A partial order R is called *total order* if for all $a, b \in A$ with $a \neq b$, either $(a, b) \in R$ or $(b, a) \in R$. For instance, R_1 above is a partial order, it is even a total order. Relations R that are orders, well, order the elements in A. If $(a, b) \in R$, then we also say that a is smaller than b with respect to R. Elements a such that there does not exist any $b \neq a$ with $(b, a) \in R$ are called *minimal*; elements a such that there does not exist any $b \neq a$ with

$(a, b) \in R$ are called *maximal*. With respect to R_1, 1 is a minimal and 3 is a maximal element. If an order is total than it has at most one minimal and at most one maximal element. (The relation $\leq$ on $\mathbb{N}$ does not have a maximal element for instance.)

A.2 Functions

A relation $f \subseteq A \times B$ such that for every $a \in A$, there is at most one $b \in B$ is called a *function*. Instead of $f \subseteq A \times B$, we often write $f : A \to B$. Let $a \in A$. If there is a b such that $(a, b) \in f$, then we usually write $f(a) = b$. ("The value of f at a is b.") Note that in this case, b is unique. If there is no such b, we say that $f(a)$ is undefined.[1] The *domain* of a function is the set

$$\mathrm{dom}(f) = \{a \in A \mid f(a) \text{ is defined}\}.$$

The *image* of f is

$$\mathrm{im}(f) = \{f(a) \mid a \in \mathrm{dom}(f)\}.$$

A function is called *total* if $\mathrm{dom}(f) = A$. If we want to stress that f need not be total, we often call it a partial function.

If $f : A \to B$ and $g : B \to C$ are total functions, then their *composition* $g \circ f$ is a function $A \to C$ defined by $(g \circ f)(a) = g(f(a))$ for all $a \in A$.

A total function $f : A \to B$ is called *surjective*, if $\mathrm{im}(f) = B$. f is called *injective* if for all $b \in B$ there is at most one a such that $f(a) = b$. f is called *bijective* if it is injective and surjective. We use the terms injective, surjective, and bijective only for total functions.

A.3 Words

Let Σ be a finite nonempty set. In the context of words, Σ is usually called an *alphabet*. The elements of Σ are called *symbols* or *letters*. A *(finite) word* w over Σ is a finite sequence of elements from Σ, i.e, it is a function $w : \{1, \ldots, \ell\} \to \Sigma$ for some $\ell \in \mathbb{N}$. ℓ is called the length of w. The length of w will also be denoted by $|w|$. There is one distinguished word of length 0, the *empty word*. We will usually denote the empty word by ε. Formally it is the sequence $\varepsilon : \emptyset \to \Sigma$.

Example A.2 *Let $u : \{1, 2, 3\} \to \{a, b, c\}$ be defined by $u(1) = a$, $u(2) = b$ and $u(3) = a$. u is a word over the alphabet $\{a, b, c\}$ of length 3.*

Let $w : \{1, \ldots, n\} \to \Sigma$ be some word. Often, we will write w in a compact form as $w(1)w(2)\ldots w(n)$ and instead of $w(i)$, we write w_i. Thus

[1] Strictly speaking, this is not quite correct, since $f(a)$ denotes the value b, which does not exists. It is more accurate to say "f is not defined at a".

we can write w in an even more compact form as $w_1 w_2 \dots w_n$. w_i is also called the ith symbol of w. The word u from the example above can be written as aba.

The set of all words of length n over Σ is denoted by Σ^n. This usually denotes the set of all n-tuples with entries from Σ, too, but this is fine, since there is a natural bijection between sequences of length n and n-tuples. The set $\bigcup_{n=0}^{\infty} \Sigma^n$ of all finite words is denoted by Σ^*. Note that $\Sigma^0 = \{\varepsilon\}$. We usually identify Σ with Σ^1, that is, letters with words of length one. Strictly speaking, these are different objects, letters are elements from a set whereas words of length one are functions $w : \{1\} \to \Sigma$. However, there is a natural bijection between these two objects mapping w to $w(1)$.

One important operation on words is concatenation. Informally, it is the word that we get when we connect two words to form a new bigger word. Formally it is defined as follows. Let $w : \{1, \dots, \ell\} \to \Sigma$ and $x : \{1, \dots, k\} \to \Sigma$ be two words. Then the concatenation of w and x is the function

$$\begin{aligned} \{1, \dots, \ell + k\} &\to \Sigma \\ i &\mapsto \begin{cases} w(i) & \text{if } 1 \leq i \leq \ell \\ x(i - \ell) & \text{if } \ell + 1 \leq i \leq \ell + k. \end{cases} \end{aligned}$$

We denote the concatenation of w and x by wx. Let $v = ca$. Then the concatenation of u from the example above and v is $abaca$. Concatenation is associative but not commutative. The empty word is a neutral element, i.e., $\varepsilon w = w\varepsilon = w$ for all words w. If we concatenate a word w with itself, we will also write w^2 instead of ww. While this does not save to much space, writing w^7 instead of $wwwwwww$ is much more impressive.

A word x is called a *subword* of y if there are words u and v such that $y = uxv$. If $u = \varepsilon$, then x is called a *prefix* of y. If $v = \varepsilon$, then x is called a *suffix* of y.

A.4 Exercises

Basic exercises

Exercise A.1 *Decide whether following relations over $\{1, 2, 3, 4\}$ are reflexive, symmetric, antisymmetric, or transitive? Which are a partial order?*

- $R = \{(1, 1), (2, 2), (3, 3), (4, 4), (1, 2), (1, 3), (1, 4)\}$
- $S = \{(1, 1), (3, 3), (4, 4), (3, 4), (4, 3)\}$

Exercise A.2 *Let Σ be an alphabet. Prove that the relation "the length of u is smaller than or equal to the length of v" is a reflexive and transitive relation on Σ^*. If $|\Sigma| = 1$, then it is even an order.*

Exercise A.3 *Let A, B, and C be sets and let $f : A \to B$ and $g : B \to C$ be total functions that are injective. Construct an injective function $A \to C$.*

The next exercise might be close to trivial, however, its statement is important when we want to do proofs by induction of the length of words.

Exercise A.4 *For word w with $|w| \geq 1$, there is a word w' with $|w'| = |w|-1$ and a letter $\sigma \in \Sigma$ (recall that we identify letters with words of length 1) such that $w = w'\sigma$. This decomposition is unique.*

Intermediate exercises

Exercise A.5 *Show that if R is an order, then R is acyclic, i.e., there does not exist a sequence $a_1, \ldots, a_i$ of pairwise distincts elements such that*

$$(a_1, a_2), (a_2, a_3), \ldots, (a_{i-1}, a_i), (a_i, a_1) \in R.$$

Exercise A.6 *Let Σ be an alphabet and let R be a total order on Σ. The lexicographic order $\leq_{\text{lex}}^R$ with respect to R on Σ^n is defined as follows:*

$$u_1u_2\ldots u_n \leq_{\text{lex}}^R v_1v_2\ldots v_n \text{ if } u_iRv_i \text{ where } i = \min\{1 \leq j \leq n \mid u_j \neq v_j\} \text{ or } i \text{ does not exist.}$$

Show that $\leq_{\text{lex}}^R$ is indeed a total order.

Exercise A.7 *1. Let A be a finite set and let $f : A \to A$ be a total function. Show that the following statements are equivalent:*

- *f is injective*
- *f is surjective*
- *f is bijective*

2. Is the previous part still valid when A is an infinite set?

Exercise A.8 *Prove that Σ^* with concatenation as an operation forms a* monoid, *that is, concatenation is associative and there is a neutral element.*

1 Finite automata and regular languages

1.1 The Chomsky hierachy

In the 1950s, Noam Chomsky [Cho56, Cho59] started to formalize (natural) languages by *generative grammars*, that is, a set of rules that describes how to generate sentences. While the purpose of Noam Chomsky was to study natural languages, his ideas turned out to be very useful in computer science. For instance, programming languages are often described by so-called context-free grammars.[1]

Chomsky studied four types of rules that led to four different types of languages, usually called type-0, type-1, type-2, and type-3. We will formally define what a grammar is when we come to type-2 languages (also called context-free languages) since grammars are natural for type-2 languages. For type-3 languages (also called regular languages), finite automata are the more natural model. But once we have characterized type-2 languages in terms of grammars, we will do so for type-3, too.

Excursus: Noam Chomsky

Noam Chomsky (born 1928 in Philadelphia, USA) is a linguist. In the 1960s, he became known outside of the scientific community for his pretty radical political views.

In computer science, he is mainly known for the study of the power of formal grammars. The so-called Chomsky hierachy contains four classes of languages that can be generated by four different kinds of grammars.

1.2 Finite automata

Finite automata describe systems that have only a finite number of states. Consider the following toy example, a coffee vending machine. This machine sells coffee for 40 cents.[2] A potential customer can perform the following actions: He can insert coins with the values 10 cents or 20 cents. Once he inserted 40 cents or more, the machine brews a coffee. If the customer has not

[1] This statement is not quite correct or even completely false, depending on your point of view. For instance, context-free grammars can describe whether loops are nested correctly, however, they cannot check whether a variable was declared previously.

[2] No, you cannot find this machine on campus.

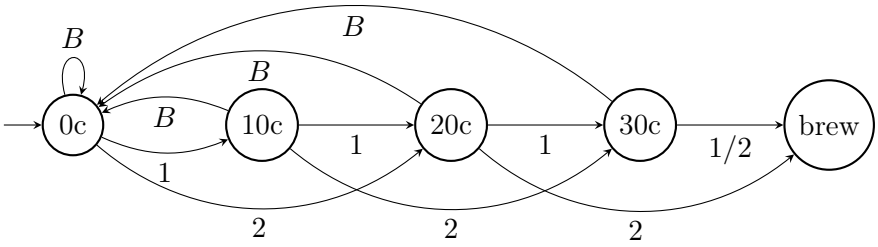


Figure 1.1: A finite automaton that models the coffee vending machine. A label of 1 or 2 on the edge means that the customer has inserted 10 or 20 cents, respectively. The notation 1/2 means that we can take this edge if either a 1 or a 2 is read. The label B means that the "Money back" button was pressed.

	1	2	B
0c	10c	20c	0c
10c	20c	30c	0c
20c	30c	brew	0c
30c	brew	brew	0c
brew	—	—	—

Figure 1.2: The COFFEE automaton written as a table. Each line corresponds to one state of the automaton. The entry "—" means that the corresponding transition is undefined.

inserted 40 cents so far, he can press the "Money back" button. The machine keeps any overpaid money. Figure 1.1 shows a diagram of the automaton COFFEE. The machine has 5 states. The four states 0c, 10c, 20c, and 30c correspond to the amount of money inserted so far, the state brew is entered if at least 40 cents have been inserted. Of course, the machine starts in the state 0c. This is indicated by the arrow on the left side of the circle. An arc from one state to another means that if the customer performs the action the edge is labeled with, then the automaton will change the state accordingly. Once the state brew is reached, the machine is supposed to brew a coffee. A clever coffee machine would then go back to the start state but we leave our machine as it is for now.

Exercise 1.1 *1. Modify the coffee automaton such that it gives change back. The amount of change should be indicated by the state that the automaton ends in.*

2. Modify the coffee automaton such that the customer has the choice

between several types of coffees.

Nowadays, finite automata still have applications. We list only three of them due to the ignorance of the author:

- Finite automata (and variants thereof) are used to verify systems that have only a finite number of states.
- Finite automata can be used for string matching, see for instance [CLRS09, Chapter 32].
- Finite automata can be used as a tokenizer to preprocess the source code of a computer program.

Formally, we define finite automata as follows:

Definition 1.1 *A finite automaton is described by a* 5*-tuple* $(Q, \Sigma, \delta, q_0, Q_{\text{acc}})$*:*

1. Q *is a finite set, the set of* states*.*
2. Σ *is a finite set, the* input alphabet*.*
3. $\delta : Q \times \Sigma \to Q$ *is the* transition function*,* δ *might be partial.*
4. $q_0 \in Q$ *is the* start state*.*
5. $Q_{\text{acc}} \subseteq Q$ *is the set of* accepting states*.*

1.2.1 Computations and regular languages

A finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ starts in state q_0. Then it reads the first symbol w_1 of its input $w = w_1 \dots w_n$ and moves from q_0 to the state $\delta(q_0, w_1)$. It continues with reading the second symbol w_2 and so on. This means that in the transition diagram, we always follow the arcs labeled by $w_1, w_2, \dots$ starting in q_0.

Definition 1.2 *Let* $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ *be a finite automaton. Let* $w \in \Sigma^*$*,* $|w| = n$*.*

1. *A sequence* $s_0, \dots, s_n \in Q$ *is called a* computation *of* M *on* w *if*
   (a) $s_0 = q_0$*,*
   (b) *for all* $0 \leq \nu < n$*:* $\delta(s_\nu, w_{\nu+1}) = s_{\nu+1}$*,*
2. *The computation is called an* accepting computation *if in addition,* $s_n \in Q_{\text{acc}}$*. Otherwise the computation is called a* rejecting computation*.*

Remark 1.3 *1. Since δ is a function, a computation, if it exists, is always unique.*

2. A computation of M on w need not exist, since $\delta(r_\nu, w_{\nu+1})$ might be undefined. We will treat such an unfinished computation also as a rejecting computation. In Lemma 1.6 below, we show that we can always assume that δ is total and that there are no unfinished computations.

Definition 1.4 *1. A finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$* accepts *a word $w \in \Sigma^*$ if there is an accepting computation of M on w. Otherwise, we say that M* rejects *w.*

2. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$ is the language that is recognized by M.

3. $A \subseteq \Sigma^*$ is a regular language if there is a finite automaton M such that $A = L(M)$.

4. REG *denotes the set of all regular languages.*

Consider our automaton COFFEE. We did not specify the accepting states of COFFEE, the natural choice is of course just to have one accepting state, namely, brew. Then COFFEE accepts a word $w \in \{1, 2, B\}^*$ if the corresponding sequence of actions results in a coffee for the customer. For instance, 22, 112, 122[3], and $12B12B12B22$[4] are accepted by COFFEE whereas 1, 12, $12B12B$ are not. L(COFFEE) is the set of all words that result in a coffee for the customer.

It will be useful to extend the transition function $\delta : Q \times \Sigma \to Q$ to words over Σ. We inductively define the *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$. We first define δ^* for words of length 0, then of length 1, length 2, and so on. For $q \in Q$, $v \in \Sigma^*$, and $\sigma \in \Sigma$, we set

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, v\sigma) = \begin{cases} \delta(\delta^*(q, v), \sigma) & \text{if } \delta^*(q, v) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

(Note that every nonempty word w can be uniquely decomposed as $w = v\sigma$ with $|v| = |w| - 1$ and $\sigma \in \Sigma$. Therefore, δ^* is well-defined.) The first line basically states that if the automaton reads the empty word (which it cannot do), then it would stay in its actual state. Next, we get $\delta^*(q, \sigma) = \delta(\delta^*(q, \varepsilon), \sigma) = \delta(q, \sigma)$ for all $q \in Q$, $\sigma \in \Sigma$. So for words of length 1, δ and δ^* coincide, which is what we want. Intuitively, $\delta^*(q, w)$ is the state that the

[3] Not all customers are smart ...

[4] Particularly funny when there is a long queue behind you ...

automaton reaches if it starts in state q and then reads w. In particular, we have:

$$M \text{ accepts } w \iff \delta^*(q_0, w) \in Q_{\text{acc}}.$$

Lemma 1.5 *For all states $q \in Q$ and all words $x, y \in \Sigma^*$,*

$$\delta^*(q, xy) = \delta^*(\delta^*(q, x), y),$$

provided that $\delta^*(q, xy)$ is defined.

Proof. The proof is by induction on $|y|$, that is, we assume that q and x can be arbitrary.
Induction base: If $y = \varepsilon$, then $\delta^*(q, x)$ is defined. We have

$$\delta^*(q, x\varepsilon) = \delta^*(q, x) = \delta^*(\delta^*(q, x), \varepsilon).$$

For the second equality, recall that $\delta^*(q, x)$ is just a state.
Induction step: For the induction step, our induction hypothesis is that for all words y of a fixed length n and for all $q \in Q$ and for all $x \in \Sigma^*$:

$$\delta^*(q, xy) = \delta^*(\delta^*(q, x), y),$$

provided that $\delta^*(q, xy)$ is defined. We have to show that this is also true for all words of the form $y\sigma$, that is, for words of length $n + 1$. Assume that $\delta^*(q, xy\sigma)$ is defined. Then $\delta^*(q, xy)$ is defined, too. We have

$$\delta^*(q, xy\sigma) = \delta(\delta^*(q, xy), \sigma) = \delta(\delta^*(\delta^*(q, x), y), \sigma) = \delta^*(\delta^*(q, x), y\sigma).$$

The second equality follows from the induction hypthesis, the other two from the definition of δ^*. ∎

1.2.2 How to design a finite automaton

How does one design a finite automaton? In the case of the coffee vending machines, the states were already implicit in the description of the problem and we just had to extract them. In other cases, this might be not this clear. Assume we want to design an automaton $M_1 = (Q, \{0, 1\}, \delta, q_0, Q_{\text{acc}})$ that accepts exactly the strings that contain three 0's in a row, i.e, $L(M_1)$ shall be $L_1 = \{w \in \{0, 1\}^* \mid 000 \text{ is a subword of } w\}$.

To put ourselves into the position of a finite automaton, which has only a finite amount of memory and can only look at one symbol at a time, think of a word with a quadrillion of symbols, too many to look at all of them at once and too many to remember them all. Now we have to scan them one by one. What we have to remember is the number of 0's that we saw after the last 1.

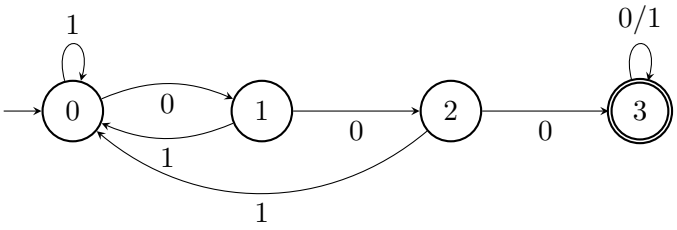


Figure 1.3: The finite automaton M_1. The double circle around state 3 indicates that 3 is an accepting state.

We have four states, 0, 1, 2, and 3. These states count the number of 0's. If we are in state i and we read another 0, then we go to state $i+1$. Once we reached state 3, we know that the word is in L_1. So $Q_{\text{acc}} = \{3\}$ and once we enter 3, we will never leave it. Whenever we read a 1 in the states 0, 1, or 2, we have to go back to state 0.

Exercise 1.2 *The automaton M_1 basically searches for the string* 000 *in the word w. Design a similar automaton that searches for the sequence* 010. *Can you devise an algorithm that given any sequence s, constructs an automaton that searches for s in a given word w?*

1.3 Closure properties, part I

To get a deeper understanding of regular languages, let's try to prove some closure properties. We start with complementation: Given $L \in \mathsf{REG}$ is $\bar{L} = \Sigma^* \setminus L$ again regular?[5] We have an automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ for L and we have to construct an automaton $\bar{M}$ for $\bar{L}$. The first idea is to exchange the accepting states and the rejecting states, that is, $Q \setminus Q_{\text{acc}}$ will be the set of accepting states of $\bar{M}$. If $w \in L(M)$, then $\bar{M}$ will indeed reject w, since the accepting computation of M on w is turned into a rejecting one of $\bar{M}$. If $w \notin L(M)$, there is a problem: A rejecting computation of M on w is turned into an accepting computation of $\bar{M}$ on w. This is fine. But $w \notin L(M)$ can also mean that there is an unfinished computation of M on w. But then the computation of $\bar{M}$ on w will also be unfinished and therefore $w \notin L(\bar{M})$. The next lemma shows how to make the transition function a total function. Once we have done this, there are no unfinished computations any more and our construction above will work.

Lemma 1.6 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a finite automaton. Then there is a finite automaton $M' = (Q', \Sigma, \delta', q_0, Q_{\text{acc}})$ such that δ' is a total function and $L(M) = L(M')$.*

[5] We always assume that the universe Σ^* of L is known, so it is clear what the complement is.

Proof overview: We add an extra *dead-lock* state to Q. Whenever δ is not defined, M' enters the dead lock state instead and can never leave this state.

Proof. Define $Q' = Q \cup \{\text{dead-lock}\}$, where dead-lock $\notin Q$. For $q \in Q$ and $\sigma \in \Sigma$, we set

$$\delta'(q,\sigma) = \begin{cases} \delta(q,\sigma) & \text{if } \delta(q,\sigma) \text{ is defined,} \\ \text{dead-lock} & \text{otherwise, and} \end{cases}$$
$$\delta'(\text{dead-lock},\sigma) = \text{dead-lock}.$$

δ' is obviously total. If $w \in \Sigma^*$ is accepted by M, then the computation of M' on w is the same as the one of M on w. Thus M' also accepts w. If w is rejected by M, then there is a rejecting computation of M on w or there is no finished computation of M at all on w. In the first case, the computation is a rejecting computation of M' on w, too. In the latter case, $\delta^*(q_0, w)$ is undefined. But this means that M' will enter dead-lock, which it cannot leave. Thus there is a rejecting computation of M' on w. ∎

Next we try to show that REG is closed under intersection and union, i.e, if $A, B \subseteq \Sigma^*$ are regular languages, so are $A \cup B$ and $A \cap B$. (We assume that A and B are languages over the same alphabet Σ. If this is not the case, we take the union of the alphabets of A and B as the new alphabet.) The construction that we will use is called *product automaton*. The product construction also works for the set difference $A \setminus B$. This in particular implies that REG is closed under complementation, since Σ^* is regular. (How does an automaton for Σ^* look like? Hint: you can do it with one state.) If M_1 and M_2 with states Q_1 and Q_2 are automata with $L(M_1) = A$ and $L(M_2) = B$, then the product automaton will have states $Q_1 \times Q_2$. This automaton can simulate M_1 and M_2 in parallel. We use the first component of the tuples to simulate M_1 and the second to simulate M_2.

Lemma 1.7 *Let $M_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, Q_{\text{acc},1})$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, Q_{\text{acc},2})$ be two finite automata such that δ_1 and δ_2 are total functions. Then the transition function Δ defined by*

$$\begin{array}{rccc} \Delta: & (Q_1 \times Q_2) \times \Sigma & \to & Q_1 \times Q_2 \\ & ((q_1, q_2), \sigma) & \mapsto & (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) \end{array}$$

fulfills

$$\Delta^*((q_1, q_2), w) = (\delta_1^*(q_1, w), \delta_2^*(q_2, w))$$

for all $q_1 \in Q_1$, $q_2 \in Q_2$, and $w \in \Sigma^*$.

Proof. The proof is by induction on $|w|$. We have $\Delta^*((q_1, q_2), \varepsilon) = (q_1, q_2)$.

For the induction step, let $w = w'\sigma$. We have

$$\begin{aligned}\Delta^*((q_1, q_2), w) &= \Delta(\Delta^*((q_1, q_2), w'), \sigma) \\ &= \Delta((\delta_1^*(q_1, w'), \delta_2^*(q_2, w')), \sigma) \\ &= (\delta_1(\delta_1^*(q_1, w'), \sigma), \delta_2(\delta_2^*(q_2, w'), \sigma)) \\ &= (\delta_1^*(q_1, w), \delta_2^*(q_2, w)).\end{aligned}$$

The second equality follows from the induction hypothesis. ∎

Theorem 1.8 REG *is closed under intersection, union, and set difference, i.e, if $A, B \subseteq \Sigma^*$ are regular languages, then $A \cap B$, $A \cup B$, and $A \setminus B$ are regular, too.*

Proof. Let $M_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, Q_{\text{acc},1})$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, Q_{\text{acc},2})$ be finite automata with $L(M_1) = A$ and $L(M_2) = B$. We may assume that δ_1 and δ_2 are total functions. Let $M = (Q_1 \times Q_2, \Sigma, \Delta, (q_{0,1}, q_{0,2}), F)$ where Δ is the function as defined in Lemma 1.7.

By defining the set of accepting states F appropriately, M will recognize $A \cap B$, $A \cup B$, or $A \setminus B$. For $A \cap B$, we set $F = Q_{\text{acc},1} \times Q_{\text{acc},2}$. By Lemma 1.7, $\Delta^*((q_{0,1}, q_{0,2}), w) = (\delta^*(q_{0,1}, w), \delta^*(q_{0,2}, w))$. We have $w \in A \cap B$ iff $\delta^*(q_{0,1}, w) \in Q_{\text{acc},1}$ and $\delta^*(q_{0,2}, w)) \in Q_{\text{acc},2}$. Thus, the choice for F above is the right one. For $A \cup B$, we set $F = Q_1 \times Q_{\text{acc},2} \cup Q_{\text{acc},1} \times Q_2$. For $A \setminus B$, we set $F = Q_{\text{acc},1} \times (Q_2 \setminus Q_{\text{acc},2})$. ∎

1.4 Exercises

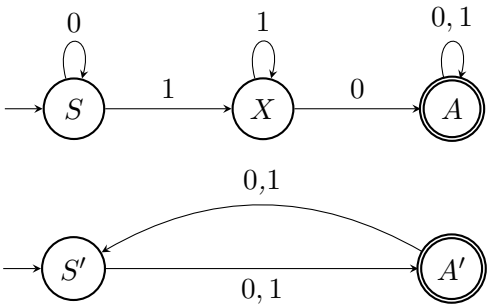
Basic exercises

Exercise 1.3 *Construct automata for the following languages.*

1. $A = \{x \in \{0,1\}^* \mid x \text{ contains at most three } 1\text{'s}\}$
2. $B = \{x \in \{0,1\}^* \mid \text{the number of } 1\text{'s in } x \text{ is divisible by } 4\}$

Exercise 1.4 *Prove that every finite language is regular.*

Exercise 1.5 *Construct the product automaton $M_1 \times M_2$ of the following two finite automata M_1 and M_2. Choose the accepting states in such a way that $x \in L(M_1 \times M_2)$ if exactly one of M_1 and M_2 accepts x ("symmetric difference").*



Intermediate exercises

Exercise 1.6 *Consider the following language*

$$C = \{x \in \{0,1\}^* \mid x \text{ interpreted as a binary number is divisible by } 3\}.$$

The least-significant bit can be the left-most bit of x or the right-most bit of x. Prove that in both cases, C is regular. To simplify matters, we allow leading 0's, so for instance, the strings $\varepsilon, 0, 00, \ldots$ are all valid encodings of the natural number 0.

Exercise 1.7 *Prove that for every regular language L, the following language is regular:*

$$L_{pf} = \{x \in L \mid \text{no proper prefix of } x \text{ is in } L\}.$$

Advanced exercises

For $x, y \in \{0,1\}^*$, we write $x \preceq y$ if x is a substring of y. A language $L \subseteq \{0,1\}^*$ is called *substring closed*, if $y \in L$ and $x \preceq y$ implies $x \in L$.

Exercise 1.8

1. *Prove the following: If $A \subseteq \{0,1\}^*$ is infinite, then there are $x, y \in A$ with $x \neq y$ and $x \preceq y$.*
2. *Prove that every substring closed language $L \subseteq \{0,1\}^\star$ is regular.*

2 Nondeterministic finite automata

In the last chapter, we showed that REG is closed under the operations complementation, union, and intersection. In this chapter, among other things, we want to show its closure under concatenation and Kleene closure.

Definition 2.1 *Let $A, B \subseteq \Sigma^*$.*

1. *The* concatenation *of A and B is*

$$AB = \{wx \mid w \in A, x \in B\}.$$

2. *The* Kleene closure *of A is*

$$A^* = \{x_1 x_2 \dots x_m \mid m \geq 0 \text{ and } x_\mu \in A,\ 1 \leq \mu \leq m\}.$$

The concatenation AB of A and B is the set of all words that consist of one word from A concatenated with one word from B. The Kleene closure is the set of all words that we get when we concatenate an arbitrary number of words (this includes zero words) from A. The concatenation of zero words is the empty word by definition.

Example 2.2 1. $\{aa, ab\}\{a, bb\} = \{aaa, aabb, aba, abbb\}$.

2. $\{aa, b\}^* = \{\varepsilon, aa, b, aaaa, aab, baa, bb, aaaaaa, aaaab, \dots\}$ *is the set of all words in which a's always occur in pairs.*

Exercise 2.1 *Prove the following:*

1. $\emptyset A = A\emptyset = \emptyset$ *for all* $A \subseteq \Sigma^*$.

2. $\emptyset^* = \{\varepsilon\}$ *and* $\{\varepsilon\}^* = \{\varepsilon\}$.

Exercise 2.2 *Prove the following:*

1. *If* $A, B \subseteq \Sigma^*$ *are finite, then* $|AB| \leq |A| \cdot |B|$.

2. *If* $A \neq \emptyset$ *and* $A \neq \{\varepsilon\}$, *then* A^* *is infinite.*

2.1 Nondeterminism

When we showed that REG is closed under union or intersection, we took two automata for $A, B \in REG$ and constructed another automaton out of these two automata for $A \cap B$ or $A \cup B$. Here is an attempt for AB: $w \in AB$ if there are $x \in A$ and $y \in B$ such that $w = xy$. So we could first run A on x and then B on y. The problem is that we do not know when we leave x and enter y. The event that A enters an accepting state is not enough; during the computation on x, A can enter and leave accepting states several times.

For instance, let $A = \{x \in \{0,1\}^* \mid \text{the number of 0's in } x \text{ is even}\}$ and $B = \{y \in \{0,1\}^* \mid \text{the number of 1's in } y \text{ is odd}\}$. How does an automata for AB look like? In a first part, we have to count the 0's modulo 2. At some point, we have to switch and count the 1's modulo 2. Figure 2.1 shows an "automaton" for AB. The part consisting of the states 0-even and 0-odd counts the 0's modulo 2. From the state 0-even, we can go to the second part of the automaton consisting of the states 1-even and 1-odd. This part counts the number of 1's modulo 2. The state 0-even is left by two arrows that are labeled with 0 and two arrows that are labeled with 1. Such an automaton is called *nondeterministic*. The automaton accepts a word if there is a sequence of choices such that the automaton ends in an accepting state. Among other things, we use nondeterminism here to construct a nondeterministic finite automaton for AB. The amazing thing is, that we can simulate a nondeterministic finite automaton by a deterministic one.

Another way to introduce nondeterminism are ε-transitions. These are arrows in the transition diagram that are labeled with ε. This means that the automaton may choose to make the ε-transition without reading a symbol of the input. Figure 2.2 shows an automaton with ε-transitions for AB. ε-transitions do not enhance the power of nondeterministic automata, but it allows us to present automata in a more compact way. In our toy example, we only save one edge, but in bigger examples, the transition diagram can become much more readable by using ε-transitions.

2.1.1 Formal definition

For a set S, $\mathcal{P}(S)$ denotes the *power set* of S, that is the set of all subsets of S. For an alphabet Σ, Σ_ε denotes the set $\Sigma \cup \{\varepsilon\}$.

Definition 2.3 *A* nondeterministic finite automaton *is described by a* 5*-tuple* $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$*:*

1. Q *is a finite set, the set of* states*.*
2. Σ *is a finite set, the* input alphabet*.*
3. $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ *is the* transition function*.*

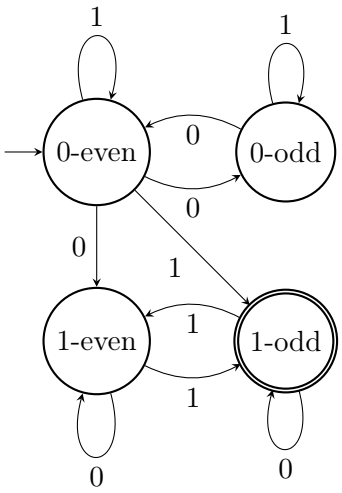


Figure 2.1: A nondeterministic finite automaton for AB.

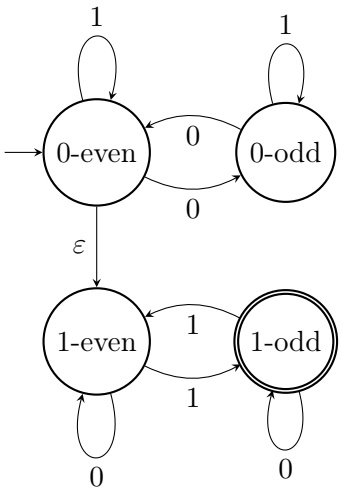


Figure 2.2: A nondeterministic finite automaton with ε-transitions for AB.

The golden rule of nondeterminism

Nondeterminism is a useful theoretical concept, but we do not not know any physical realisation of it.[a]

Nondeterminism is interesting because it is useful tool: We can model different choices. We do not know how to build an nondeterministic finite automaton in reality.

[a] Maybe the NSA does. Hopefully not ...

4. $q_0 \in Q$ *is the* start state.

5. $Q_{\text{acc}} \subseteq Q$ *is the set of* accepting states.

If δ is only a function $Q \times \Sigma \to \mathcal{P}(Q)$, then M is called a nondeterministic finite automaton without ε-transitions.

To distinguish between nondeterministic finite automata and the finite automata of the last chapter, we call the latter ones *deterministic finite automata*. Deterministic finite automata are a special case of nondeterministic finite automata. Formally, this is not exactly true. But from the transition function $\delta : Q \times \Sigma \to Q$ of a deterministic finite automata, we get a transition function $Q \times \Sigma \to \mathcal{P}(Q)$ by $(q, \sigma) \mapsto \{\delta(q, \sigma)\}$ for all $q \in Q, \sigma \in \Sigma$. This means that the transition function now does not map to states but to sets consisting of a single state.

2.1.2 Computations and computation trees

A nondeterministic finite automaton starts in q_0. Then it has several choices. There is the possibility to make an ε-transition and enter a new state without reading a symbol. Or it may read a symbol and go to one of several states. On one word w, there may be plenty of computations now.

Definition 2.4 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be nondeterministic finite automaton. Let $w \in \Sigma^*$, $w = w_1 \dots w_n$.*

1. *$s_0, s_1, \dots, s_m \in Q$ is called a* computation *of M on w if we can write $w = u_1 u_2 \dots u_m$[1] with $u_\mu \in \Sigma_\varepsilon$ such that*

   (a) *$s_0 = q_0$.*

   (b) *for all $0 \leq \mu < m$, $s_{\mu+1} \in \delta(s_\mu, u_{\mu+1})$.*

[1] m can be larger than n. In this case, there are indices $i_1, \dots, i_n \in \{1, \dots, m\}$ such that $u_{i_1} \dots u_{i_n} = w$. All other $u_j = \varepsilon$.

2. *The computation above is called an* accepting computation *if* $s_m \in Q_{\text{acc}}$. *Otherwise, it is called a* rejecting computation.

Recall that $xy = x\varepsilon y$ for any two words x, y. In the decomposition $w = u_1 u_2 \dots u_m$ above, we have inserted ε at the the places where M wants to make an ε-transition.

Definition 2.5 1. *A nondeterministic finite automaton* M accepts *a word* w *if there is an accepting computation of* M *on* w. *Otherwise,* M rejects w.

2. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$ *is the language recognized by* M.

Note that in the definition above, there needs to be at least one accepting computation for M accepting w. There could be more, and there could be rejecting as well as unfinished computations, too.

We do not define nondeterministic regular languages since we will show below that for every nondeterministic finite automaton there is a deterministic one that recognizes the same language; a really surprising result.

Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$. Again, we would also like to extend the transition function δ to words in Σ^* as we did for deterministic finite automata. But this is not as easy. We first define the ε-closure of the transition function. For $q \in Q$ and $\sigma \in \Sigma$, $\delta^{(\varepsilon)}(q, \sigma)$ denotes all states that we can reach from q by making an arbitrary number of ε-transitions and then one transition that is labeled with σ. (And we are *not* allowed to make any ε-transitions afterwards.) Formally,

$$\begin{array}{rccl} \delta^{(\varepsilon)}: & Q \times \Sigma & \to & \mathcal{P}(Q) \\ & (q, \sigma) & \mapsto & \{r \mid \text{ there are } k \geq 0 \text{ and } s_0 = q, s_1, \dots, s_k \text{ such that} \\ & & & \quad s_{\kappa+1} \in \delta(s_\kappa, \varepsilon),\ 0 \leq \kappa < k, \text{ and } r \in \delta(s_k, \sigma)\}. \end{array}$$

For a subset $R \subseteq Q$ of the states, $R^{(\varepsilon)}$ denotes all the states in Q *from which* we can reach a state in R just by ε-transitions. Formally,

$$\begin{array}{rl} R^{(\varepsilon)} = \{r \in Q \mid & \text{there are } k \geq 0 \text{ and } s_0 = r, s_1, \dots s_k \text{ such that} \\ & s_{\kappa+1} \in \delta(s_\kappa, \varepsilon),\ 0 \leq \kappa < k, \text{ and } s_k \in R.\}. \end{array}$$

Lemma 2.6 *If* $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ *is a nondeterministic finite automaton, then* $M' = (Q, \Sigma, \delta^{(\varepsilon)}, q_0, Q_{\text{acc}}^{(\varepsilon)})$ *is a nondeterministic finite automaton without* ε*-transitions such that* $L(M) = L(M')$.

Proof. M' obviously does not have ε-transitions. Assume that M accepts w. Then we can write $w = u_1 u_2 \dots u_m$ with $u_\mu \in \Sigma_\varepsilon$, and there are states $s_0, s_1, \dots, s_m$ with $s_0 = q_0$, $s_{\mu+1} \in \delta(s_\mu, u_\mu)$ for $0 \leq \mu < m$, and $s_m \in Q_{\text{acc}}$. Let $i_1, \dots, i_n$ be indices such that $u_{i_\nu} \in \Sigma$ for $1 \leq \nu \leq n$, i.e., u_{i_ν} is w_ν, the

νth symbols of w. Let $i_0 = 0$. Then $\delta^{(\varepsilon)}(s_{i_\nu}, u_{i_{\nu+1}}) = s_{i_{\nu+1}}$ for $0 \leq \nu < n$ and $s_{i_n} \in Q_{\text{acc}}^{(\varepsilon)}$ by the construction of $\delta^{(\varepsilon)}$ and $Q_{\text{acc}}^{(\varepsilon)}$. Thus M' accepts w, too.

Conversely, if M' accepts $w = w_1 w_2 \dots w_n$ with $w_\nu \in \Sigma$ for $1 \leq \nu \leq n$, then M also accepts w: There are states $s_0, \dots, s_n$ such that $s_0 = q_0$ and $\delta^{(\varepsilon)}(s_\nu, w_{\nu+1}) = s_{\nu+1}$ for $0 \leq \nu < n$, and $s_n \in Q_{\text{acc}}^{(\varepsilon)}$. For all $0 \leq \nu < n$, there are states $s_{\nu,0} = s_\nu, s_{\nu,1}, \dots, s_{\nu,j_\nu}$ such that $\delta(s_{\nu,h}, \varepsilon) = s_{\nu,h+1}$, $0 \leq h < j_\nu$, and $\delta(s_{\nu,j_\nu}, w_{\nu+1}) = s_{\nu+1}$ by the construction of $\delta^{(\varepsilon)}$. And there are states $s_{n,0} = s_n, s_{n,1}, \dots, s_{n,j_n}$ such that $\delta(s_{n,h}, \varepsilon) = s_{n,h+1}$, $0 \leq h < j_n$, and $s_{n,j_n} \in Q_{\text{acc}}$ by the construction of $Q_{\text{acc}}^{(\varepsilon)}$. All these intermediate states together form an accepting computation of M on w. ∎

Remark 2.7 *All finished computations of a finite automaton without ε-transitions on a given input have the same length. Their number is bounded by $|Q|^n$.*

For a finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$, the extended transition function $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ is defined inductively as follows:

$$\begin{array}{rcll} \delta^*(q, \varepsilon) & = & \{q\} & \text{for all } q \in Q, \\ \delta^*(q, w\sigma) & = & \bigcup_{r \in \delta^*(q,w)} \delta^{(\varepsilon)}(r, \sigma) & \text{for all } q \in Q,\ w \in \Sigma^*,\ \sigma \in \Sigma. \end{array}$$

$\delta^*(q, x)$ are all the states that we can reach if we start from q, read x, and do not allow any ε-transition after reading the last symbol of x. Similar deterministic finite automata, we have

$$M \text{ accepts a word } x \iff \delta^*(q_0, x) \cap Q_{\text{acc}}^{(\varepsilon)}.$$

The righthand side simply means that we can reach a state by reading x from which we can reach an accepting state by ε-transitions.

It might be a little unsatisfying that δ^* maps states to sets of states. We can extend δ^* to a function $\mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ by simply setting

$$\delta^*(R, x) = \bigcup_{r \in R} \delta^*(r, x).$$

Strictly speaking, the functions δ^* on each side are different functions. However, by identifying a state r with the set $\{r\}$ of size one, we can embed the domain $Q \times \Sigma^*$ into $\mathcal{P}(Q) \times \Sigma^*$.

It is sometimes easier to represent the computations of a nondeterministic finite automaton M by a *computation tree*. We only do this for ε-transition free automata here. The computation tree of M on a word w is a rooted tree whose nodes are labeled. The root is labeled with q_0. For all states $s \in \delta(q_0, w_1)$, we add one child to q_0. This child is labeled with s. Consider one such child and let its label be s_0. For each state $r \in \delta(s_0, w_2)$, we add

one child to s_0 with label r and so forth. Even if there is already a node with the label r, we insert a new child, so one label can (and will) appear several times. The depth of the tree is bounded by n. The states of every level i of the computation tree are precisely $\delta^*(q_0, w_{\leq i})$ where $w_{\leq i}$ is the prefix of w of length i. Every path of length n from the root to a leaf corresponds to a finished computation of M on w and for every finished computation of M on w, there is such a path. M accepts w if and only if there is a path of length n from the root to some leaf that is labeled by an accepting state in the computation tree. Figure 2.3 shows the computation tree of the automaton from Figure 2.1 on the word 01011.

2.2 Determinism versus nondeterminism

Theorem 2.8 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a nondeterministic finite automaton. Then there is a deterministic finite automaton $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q}_0, \hat{Q}_{\text{acc}})$ such that $L(M) = L(\hat{M})$.*

Proof overview: By Lemma 2.6, we can assume that M does not have any ε-transitions. Look at the computation tree of M on some word w of length n. Each level i of the tree contains all the states that M can reach after reading the first i symbols of w. If the nth level contains a state from Q_{acc}, then M accepts w. A deterministic automaton has to keep track of all the states that are in one level. But it has to store them in one state. The solution is to take $\mathcal{P}(Q)$, the power set of Q, as $\hat{Q}$, the set of states of $\hat{M}$.

Proof. By Lemma 2.6, we can assume that M does not have any ε-transitions. We set $\hat{Q} = \mathcal{P}(Q)$ and $\hat{q}_0 = \{q_0\}$. We define the transition function $\hat{\delta}$ as follows:

$$\hat{\delta}(R, \sigma) = \bigcup_{r \in R} \delta(r, \sigma) \qquad \text{for all } R \in \hat{Q} \text{ and } \sigma \in \Sigma.$$

(Note that δ is a function $\hat{Q} \times \Sigma \to \hat{Q}$. It maps into the power set of Q but not into the power set of $\hat{Q}$. Thus $\hat{M}$ is deterministic!) Finally, we set $\hat{Q}_{\text{acc}} = \{R \in \hat{Q} \mid R \cap Q_{\text{acc}} \neq \emptyset\}$.

We now show by induction in the length of $w \in \Sigma^*$ that for all $R \subseteq Q$ and $w \in \Sigma^*$,

$$\hat{\delta}^*(R, w) = \bigcup_{r \in R} \delta^*(r, w).$$

For $w = \varepsilon$, $\hat{\delta}^*(R, w) = R = \bigcup_{r \in R} \delta^*(r, w)$. Now let $w = x\sigma$ with $\sigma \in \Sigma$. We

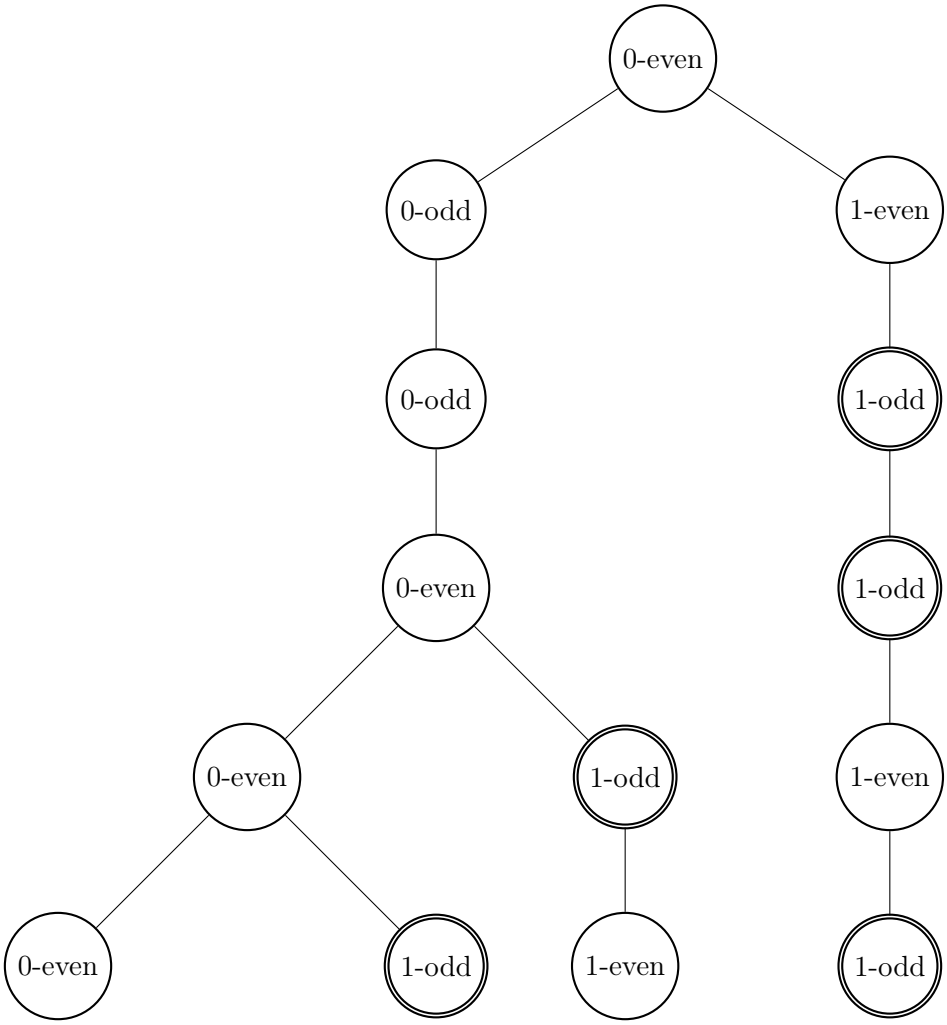


Figure 2.3: The computation tree of the automaton from Figure 2.1 on the word 01011. The root is the start state 0-even. Then the automaton reads a 0. It has two possibilities: either it moves to 0-odd or to 1-even. These two possibilities are represented by the two children. There are four different computations on 01011, two of them are accepting. These two accepting configurations correspond to splitting 01011 either as $\varepsilon \in A$ and $01011 \in B$ or $0101 \in A$ and $1 \in B$.

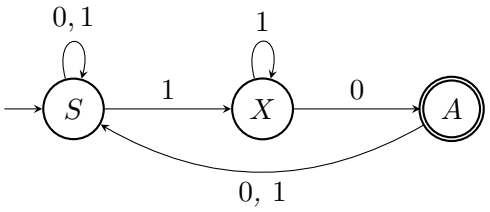
have

$$\begin{aligned}\hat{\delta}^*(R,w) &= \hat{\delta}(\hat{\delta}^*(R,x),\sigma)\\ &= \hat{\delta}(\bigcup_{r\in R}\delta^*(r,x),\sigma)\\ &= \bigcup_{s\in\bigcup_{r\in R}\delta^*(r,x)}\delta(s,\sigma)\\ &= \bigcup_{r\in R}\bigcup_{s\in\delta^*(r,x)}\delta(s,\sigma)\\ &= \bigcup_{r\in R}\delta^*(r,w)\end{aligned}$$

Above, the second equality follows from the induction hypothesis and the third equality follows from the definition of $\hat{\delta}$. For the fourth equality, note that the union in the third line and the fourth line is over the same set of s. The last equatily follows from the definition of δ^* for nondeterministic finite automata. Note that since we removed ε-transitions first, $\delta^{(\varepsilon)} = \delta$.

$\hat{M}$ accepts w iff $\hat{\delta}^*(\hat{q}_0, w) \in \hat{Q}_{\text{acc}}$. M accepts w iff $\delta^*(q_0, w) \cap Q_{\text{acc}} \neq \emptyset$. From the definition of $\hat{Q}_{\text{acc}}$ it follows that $\hat{M}$ accepts w iff M accepts w. Thus $L(M) = L(M')$. ■

Exercise 2.3 *Apply the power set construction to the following automaton:*



Which language does the automaton recognize?

Somehow one feels cheated when seeing the subset construction for the first time, but it is correct. The deterministic finite automaton pays for being deterministic by a huge increase in the number of states. If the nondeterministic automaton has n states, the deterministic automaton that we construct has 2^n states and there are examples where this is (almost) neccessary. Here is one such example.

Example 2.9 *For $n \in \mathbb{N}$, consider the language*

$$S_n = \{w \in \{0,1\}^* \mid \textit{the nth symbol from the end is } 0\}.$$

There is a nondeterministic finite automaton for S_n that has $n+1$ states, see Figure 2.4 for $n = 3$. A deterministic finite automaton could store the

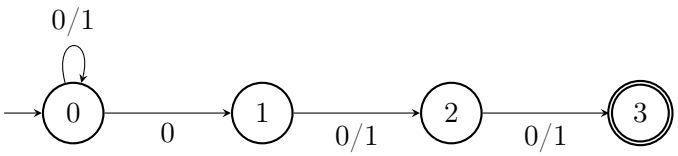


Figure 2.4: A nondeterministic finite automaton for S_3.

last n symbols that it has seen, i.e., $Q = \{0,1\}^n$. (Yes, states can also just be strings. We just have to use a finite set.) The transition function is defined by $\delta(\sigma_1 \dots \sigma_n, \tau) = \sigma_2 \dots \sigma_n \tau$ for all $\sigma_1 \dots \sigma_n \in Q$ and $\tau \in \Sigma$. The start state is 1^n and the accepting states are all states of the form $0\sigma_2 \dots \sigma_n$. This automaton has 2^n states. We will see soon that this is neccessary.

2.3 Closure properties, part II

Theorem 2.10 REG *is closed under concatenation and Kleene closure, i.e., for all $A, B \in$ REG, we have $AB, A^* \in$ REG.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a deterministic finite automaton for A and $M' = (Q', \Sigma, \delta', q_0', Q_{\text{acc}}')$ be a deterministic finite automaton for B. W.l.o.g we can assume that $Q \cap Q' = \emptyset$.

For AB, we construct an automaton $N_1 = (Q \cup Q', \Sigma, \gamma_1, q_0, Q_{\text{acc}}')$. We connect each accepting state of M by an ε-transition to the start state q_0' of M'. Therefore, N_1 will be nondeterministic. Formally,

$$\gamma_1(q,\sigma) = \begin{cases} \{\delta(q,\sigma)\} & \text{if } q \in Q \text{ and } \sigma \neq \varepsilon, \\ \{\delta'(q,\sigma)\} & \text{if } q \in Q' \text{ and } \sigma \neq \varepsilon, \\ \{q_0'\} & \text{if } q \in Q_{\text{acc}} \text{ and } \sigma = \varepsilon, \\ \text{undef.} & \text{otherwise.} \end{cases}$$

See also Figure 2.5. It is clear from the construction that $L(N_1) = AB$: Let $x \in A$ and $s_0, \dots, s_m$ be an accepting computation of M on x. Furthermore, let $y \in B$ and $s_0', \dots, s_n'$ be an accepting computation of M' on y. Then $s_0, \dots, s_m, s_0', \dots, s_n'$ is an accepting computation of N_1 on xy. Note that $s_0 = q_0$, $s_m \in Q_{\text{acc}}$, $s_0' = q_0'$, and $s_n' \in Q_{\text{acc}}'$. By construction $\gamma_1^*(q_0, x) = s_m$, $\gamma_1(s_m, \varepsilon) = q_0'$ and $\gamma_1^*(q_0', y) = s_n' \in Q_{\text{acc}}'$. Therefore, $xy \in L(N_1)$. Since x and y were arbitray, $AB \subseteq L(N_1)$. On the other hand, every accepting computation of N_1 can be split into an accepting computation of M and M'. Thus we can conclude in a similar way that $L(N_1) \subseteq AB$.

For A^*, we construct an automaton $N_2 = (Q \cup \{r\}, \Sigma, \gamma_2, r, Q_{\text{acc}} \cup \{r\})$ where $r \notin Q$. The first idea is to insert an ε-transition from every accepting

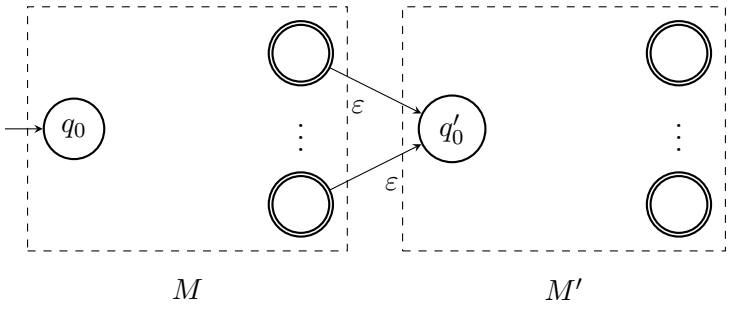


Figure 2.5: A schematic drawing of the automaton N_1.

state to the start state. In this way, whenever we enter an accepting state, N_2 can either stop or add another word from A. This automaton recognizes $A^+ = A^* \setminus \{\varepsilon\}$. To add ε to the language, one idea would be to make the start state an accepting state, too. While this adds ε to the language, this does not work (cf. Exercise 2.4). Instead, we add a new start state r that is an accepting state. This adds ε to the language. From r, there is an ε-transition to the old start state q_0 of M. Formally,

$$\gamma_2(q,\sigma) = \begin{cases} \delta(q,\sigma) & \text{if } q \in Q \text{ and } \sigma \neq \varepsilon, \\ q_0 & \text{if } q \in \{r\} \cup Q_{\text{acc}} \text{ and } \sigma = \varepsilon, \\ \text{undef.} & \text{otherwise.} \end{cases}$$

See Figure 2.6 for a schematic drawing of the construction. It is clear from the construction that $L(N_2) = A^*$. The formal proof is similar to the first part. ■

Remark 2.11 *For simplicity, we assumed that M and M' in the constructions above are deterministic. The same construction also works for nondeterministic automata.*

Exercise 2.4 *Give an example that in the proof above, we need the extra state r in N_2, i.e., show that it is not correct to create an ε-transition from each accepting state to the start state and make the start state an accepting state, too.*

2.4 Exercises

Basic exercises

Exercise 2.5 *Prove that the following languages are regular:*

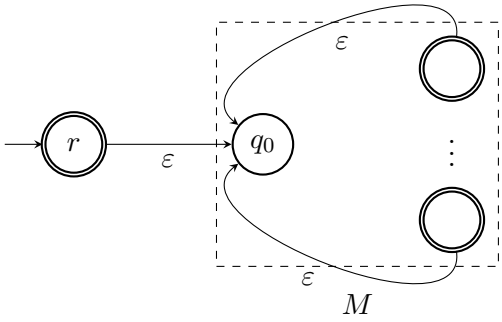
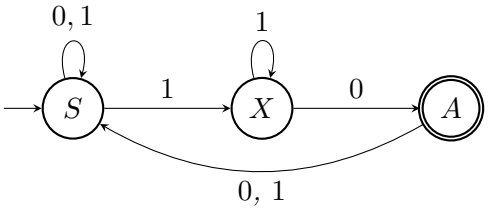


Figure 2.6: A schematic drawing of the automaton N_2.

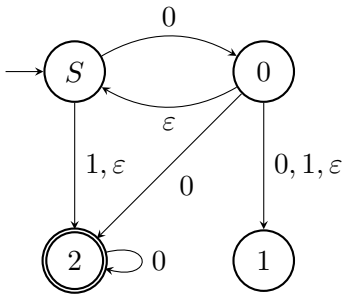
1. $A = \{x \in \{0,1\}^* \mid x$ *starts with* 00 *or ends with* $11\}$,
2. $B = \{x \in \{0,1\}^* \mid x = x_1 x_2 \ldots x_\ell,\ \ell \geq 4,\ x_1 = x_{\ell-1}$ *und* $x_2 = x_\ell\}$. *(The first two symbols are the same as the last two symbols.)*

Exercise 2.6 *Construct the computation trees of the following automaton on* ε, 101, *and* 01010*:*



Which language is recognized by the automaton?

Exercise 2.7 *Construct a nondeterministic finite automaton without* ε*-transitions equivalent to the automaton below. Which language does the automaton recognize?*



Intermediate exercises

Exercise 2.8 *Prove that for every regular language L, the following languages are regular:*

1. $L_{\text{rev}} = \{x^{\text{rev}} \mid x \in L\}$.
   Here for a word $x = x_1 x_2 \dots x_\ell$, $x^{\text{rev}} := x_\ell x_{\ell-1} \dots x_1$.
2. $L^{-0} = \{xy \mid x0y \in L\}$.
3. $L_{rot} = \{x = x_1 \dots x_\ell \mid \exists k \leq \ell : x_{k+1} \dots x_\ell x_1 \dots x_k \in L\}$.

Exercise 2.9 *In Exercise 1.6, we looked at the set of all binary string that represents a number when divisible by 3. In the solution, we obtained an automaton with three states in the case when the left-most bit was the most-significant bit. Call this language S. What is S_{rev}? What happens to the automaton for S when you convert it into an automaton for S_{rev}?*

Exercise 2.10 *A* homomorphism *is a mapping $h : \Sigma^\star \to \Sigma^\star$ such that $h(xy) = h(x)h(y)$ for all $x, y \in \Sigma^\star$.*

1. *Prove that every homomorphism fulfills $h(\varepsilon) = \varepsilon$. Prove that h is uniquely determined by the values $h(\sigma)$ for $\sigma \in \Sigma$.*
2. *Let $L \subseteq \Sigma^\star$ be regular. Prove that $h(L) = \{h(x) \mid x \in L\}$ is regular, too.*
3. *Let $L \subseteq \Sigma^\star$ be regular. Prove that $h^{-1}(L) = \{x \mid h(x) \in L\}$ is regular, too.*

Advanced exercises

Exercise 2.11 *Let L be a regular language, prove that the following language*

$$L_H = \{x \mid \exists y : xy \in L \text{ and } |x| = |y|\}$$

is regular, too.

More general, a function f is *regularity preserving* if for all regular L, the language $\{x \mid \exists y : xy \in L \text{ and } f(|x|) = |y|\}$ is regular, too. The previous exercise shows that the identity function is regularity preserving.

A set $U \subseteq \mathbb{N}$ is called *ultimately periodic*, if there are numbers N and p such that for all $n \geq N$, $n \in U$ iff $n + p \in U$. The number p is called a *period*.

Seiferas and McNaughton [SM76] prove the following theorem:

Theorem 2.12 *A function $f : \mathbb{N} \to \mathbb{N}$ is regularity preserving iff for all ultimately periodic U, $f^{-1}(U) = \{u \mid f(u) \in U\}$ is ultimately periodic, too.*

Exercise 2.12 *1. Prove the following: If M is a determinisitic finite automaton with k states over an alphabet Σ with $L(M) \neq \emptyset, \Sigma^*$, then there are strings $x \in L(M)$ and $y \in \Sigma^\star \setminus L(M)$ with $|x| < k$ and $|y| \leq k$.*

2. Prove that if M is nondeterminisic, then there is a string $x \in L(M)$ mit $|x| < k$.

3. Construct an nondeterministic finite automaton M with k states over the alphabet $\{1\}$ such that the shortest string $y \in \{1\}^\star \setminus L(M)$ has length at least $k + 1$.

Hint: $k = 8$ works.

4. Generalize the construction of part 3 and construct nondeterministic finite automata with an arbitrarily large number of states k such that the length of the shortest string in $\{1\}^\star \setminus L(M)$ is superpolynomial in k.

Hint: Chinese remainder theorem

3 Regular expressions

3.1 Formal definition

3.1.1 Syntax

Definition 3.1 *Let Σ be an alphabet and assume that the symbols "(", ")", "$\underline{\emptyset}$", "$\underline{\varepsilon}$", "+", and "*" do not belong to Σ. Regular expressions over Σ are defined inductively:*

1. *The symbols $\underline{\emptyset}$ and $\underline{\varepsilon}$ are regular expressions.*
2. *For each $\sigma \in \Sigma$, σ is a regular expression.*
3. *If E and F are regular expressions, then $(E + F)$, (EF) and (E^*) are regular expressions.*

Above $\underline{\emptyset}$ and $\underline{\varepsilon}$ are symbols that will represent the empty set and the set $\{\varepsilon\}$, but they *are not* the empty set or the empty word themselves. But since these underlined symbols usually look awkward, we will write $\emptyset$ instead of $\underline{\emptyset}$ and ε instead of $\underline{\varepsilon}$. I suppose that the reader is old enough to deal with this. It is usually clear from the context whether we mean the symbols for the empty set and the empty word or the objects themselves.

3.1.2 Semantics

Definition 3.2 *Let E be a regular expression. The language $L(E)$ described by E is defined inductively:*

1. *If $E = \emptyset$, then $L(E) = \emptyset$.*
   If $E = \varepsilon$, then $L(E) = \{\varepsilon\}$.
2. *If $E = \sigma$ for some $\sigma \in \Sigma$, then $L(E) = \{\sigma\}$.*
3. *If $E = (E_1 + E_2)$, then $L(E) = L(E_1) \cup L(E_2)$.*
   If $E = (E_1E_2)$, then $L(E) = L(E_1)L(E_2)$.
   If $E = (E_1^*)$, then $L(E) = L(E_1)^*$.

The symbol $\emptyset$ represents the empty set and the symbol ε represents the set that contains solely the empty word. A symbol $\sigma \in \Sigma$ represents the set that contains the symbol itself. These three cases form the basis of the definition. Next come the cases where E is composed of smaller expressions. The operator "+" corresponds to the union of the corresponding languages,

the concatenation of the expression corresponds to the concatenation of the corresponding languages and the "*"-operator stand for the Kleene closure. Union, concatenation, and Kleene closure are also called the *regular operations*.

Why did we put all these brackets around the expressions in the inductive definition? Well, this is to ensure that $L(E)$ is well-defined. The brackets ensure that there is only one way how to write E as a composition of one or two smaller regular expressions. (These brackets encode a so-called derivation tree, a concept that we will study in the fourth part.) For instance, if there were no brackets, then we could write $L(EFG) = L(E)L(FG)$ or $L(EFG) = L(EF)L(G)$ according to the above definition and it is not clear whether the righthand sides are the same. In fact they are, but this needs a proof, given in the next section. However, with brackets, it is clear which of the two righthand sides we mean.

Excursus: Regular expressions in Unix

Unix uses regular expressions though with a different notation and some extensions (which are just syntactic sugar):

- The dot "`.`" can mean any one symbol. This can be replaced by $\sigma_1 + \sigma_2 + \cdots + \sigma_\ell$, if $\Sigma = \{\sigma_1, \ldots, \sigma_\ell\}$. But the dot is very handy, since the alphabet in Unix is pretty large.
- `[`$\tau_1\tau_2 \ldots \tau_k$`]` is the union of the k symbols, i.e, in our notation it is $\tau_1 + \tau_2 + \cdots + \tau_k$.
- Since the ASCII symbols are ordered, one can also write something like `[a-z]`, which means all symbols between `a` and `z`.
  (Warning: this is not really true any more since the order nowadays depends on the locale. So maybe fancy characters like `ä` could occur between `a` and `z`. Therefore, shortcuts like `[:lower:]` (all lower case letters) have been introduced.)
- The union of two expressions is denoted by `|`.
- $E*$ is the Kleene closure of E, $E+$ stands for $E^+ := EE^*$ (that is, the concatenation of any positive number of words from $L(E)$), and E{`m`} stands for $\underbrace{EE \ldots E}_{m \text{ times}}$.

For instance, elements of programming languages, so-called *tokens* are usually elements from a regular language.[1]

The tool `lex` performs the so-called *lexical analysis*. Basically, it gives us the tokens from the ASCII text. We give `lex` a list of regular expressions together with an action to be performed and then `lex` scans the source code for occurences

[1] But notice that the languages of all valid source codes are usually not regular. Simple languages, like the WHILE language, which we will introduce in the next part, are usually context-free (type-2 in the Chomsky hierarchy), a concept that we will meet at the end of this book. More overloaded ones like C++ or JAVA usually are even more complicated to describe.

of elements of the regular languages. Whenever one is found, it executes the corresponding action (like inserting a new entry in the list of used variables, etc.).

3.2 Algebraic laws

Like addition and multiplication over, say, the integers fulfill several algebraic laws, the regular operations fulfill an abundance of such laws, too. If we write $17 + 4 = 21$, then this means that the natural numbers on the left-hand side and the right-hand side are the same; though the words on both side are not: the left-hand side is the concatenation of the four symbols 1, 7, +, and 4, the right hand side is the concatenation of 2 and 1. In the same way, if we write $E = F$ for two regular expressions, then this means that $L(E) = L(F)$, i.e., the two expressions describe the same language. Like $x + y = y + x$ holds for all natural numbers x and y, we can formulate such laws for regular expressions.

Theorem 3.3 *For all regular expressions E, F, and G, we have*

1. $E + F = F + E$ *(commutativity law for union),*
2. $(E + F) + G = E + (F + G)$ *(associativity law for union),*
3. $(EF)G = E(FG)$ *(associativity law for concatenation),*
4. $\emptyset + E = E + \emptyset = E$ *($\emptyset$ is an identity for union),*
5. $\varepsilon E = E\varepsilon = E$ *(ε is the identity for concatenation),*
6. $\emptyset E = E\emptyset = \emptyset$ *($\emptyset$ is an annihilator for concatenation),*
7. $E + E = E$ *(union is idempotent),*
8. $(E + F)G = (EG) + (FG)$ *(right distributive law),*
9. $E(F + G) = (EF) + (EG)$ *(left distributive law),*
10. $(E^*)^* = E^*$,
11. $\emptyset^* = \varepsilon$,
12. $\varepsilon^* = \varepsilon$.

Proof. We only prove the first, sixth, and tenth item. The rest is left as an exercise.

For the first item, we use the fact that the union of sets is commutative: $L(E + F) = L(E) \cup L(F) = L(F) \cup L(E) = L(F + E)$.

Item 6: For any two languages A and B, AB is the set of all words $w = ab$ with $a \in A$ and $b \in B$. If one of A and B is empty, then no such word w exists, thus $AB = \emptyset$ in this case.

Item 10: Let $L := L(E)$. $L^* \subseteq (L^*)^*$ is clear, since $(L^*)^*$ is the set of all words that we get by concatenating an arbitrary number of words from L^*, in particular of one word. To show that $(L^*)^* \subseteq L^*$, let $x_1, \ldots, x_n \in L^*$. This means that we can write every x_ν as $y_{\nu,1} \ldots y_{\nu,j_\nu}$ with $y_{\nu,i} \in L$ for $1 \leq i \leq j_\nu$, $1 \leq \nu \leq n$. This means that we can write $x_1 \ldots x_n = y_{1,1} \ldots y_{1,j_1} y_{2,1} \ldots y_{n-1,j_{n-1}} y_{n,1} \ldots y_{n,j_n} \in L^*$. ■

Exercise 3.1 *1. Prove the remaining parts of Theorem 3.3.*

2. Construct two regular expressions E and F with $EF \neq FE$.

Exercise 3.2 *Prove the following identities:*

1. $(E + F)^* = (E^* F^*)^*$.
2. $\varepsilon + EE^* = E^*$.
3. $(\varepsilon + E)^* = E^*$.

3.2.1 Precedence of operators

In the definition of regular expressions, whenever we built a new expression out of two other, we placed brackets around the expression. Since these brackets often make the expressions hard to read, we would like to omit some of them. Note that the algebraic laws proven above help us: For instance, since concatenation is associative, that is, $(EF)G = E(FG)$, we can simply write EFG instead. $L(EFG)$ is still well-defined: There are now two ways how to compute $L(EFG)$, corresponding to the two expressions $(EF)G$ and $E(FG)$, but both ways will give the same results. The same can be done with union.

We can save some more brackets by making conventions about the order of precedence of operators:

- Kleene closure * has the highest precedence. That is, it applies to the smallest sequence of symbols to its left that form a valid regular expression.
- Next comes concatenation.
- Union has the lowest priority.

This is just like the precedence of exponentiation over multiplication over addition in arithmetic expressions.

We can always go back to an expression that is fully bracketed, however, not in a unique way. We make the convention to group concatenation and unions from left to right, but any other choice is fine, too.

Example 3.4 *Consider the expression* $01^*0 + 0 + \varepsilon$. *It is transformed back into* $((((0(1^*))0) + 0) + \varepsilon)$.

3.3 Regular expressions characterize regular languages

As the name suggests, regular expression characterize—surprise, surprise!—exactly the regular languages. This means that for every regular expression E, $L(E)$ is regular. And conversely, if L is regular, then we can find a regular expression E such that $L(E) = L$.

Theorem 3.5 *If E is a regular expression, then $L(E) \in \mathsf{REG}$.*

Proof overview: The proof is done by structural induction. We first prove the statement of the theorem for the simple regular expressions (Definition 3.1, 1. and 2.). This is the induction basis. In the induction step, we have prove the statement for the expressions $E_1 + E_2$, E_1E_2, and E_1^* and the induction hypothesis is that $L(E_1)$ and $L(E_2)$ are regular. We can reduce structural induction to "regular" induction by viewing it as induction in the number of applications of the operators "+", concatenation, and "*".

Proof. Induction base: We have to construct automata that accept the languages $\emptyset$, $\{\varepsilon\}$, and $\{\sigma\}$. But this is easy (see also Exercise 3.3).
Induction step: We are given $E = E_1 + E_2$, $E = E_1E_2$, or $E = E_1^*$ and we know that $L(E_1)$ and $L(E_2)$ are regular. We have to show that $L(E) = L(E_1) \cup L(E_2)$ or $L(E) = L(E_1)L(E_2)$ or $L(E) = L(E_1)^*$, respectively, is regular. But this is clear, since we already showed that REG is closed under union, concatenation and Kleene closure. ∎

Exercise 3.3 *Construct finite automata that accept the languages $\emptyset$, $\{\varepsilon\}$, and $\{\sigma\}$ for $\sigma \in \Sigma$.*

Theorem 3.6 *For every deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$, there is a regular expression E such that $L(M) = L(E)$.*

Proof overview: We assume that $Q = \{1, 2, \ldots, n\}$. Assume that 1 is the start state and $j_1, \ldots, j_\ell$ are the accepting states. For each pair i, j of states, we will inductively define expressions $E_{i,j}^k$, $0 \leq k \leq n$, such that in the end, $L(E_{i,j}^n)$ is exactly the set of all strings w such that $\delta^*(i, w) = j$, i.e, if M starts in i and reads w, then it ends in j. But then $L(M) = L(E_{1,j_1}^n + \cdots + E_{1,j_\ell}^n)$. What does the superscript k mean? $L(E_{i,j}^k)$ will be the set of all words such that $\delta^*(i, w) = j$ and for each prefix w' of w with $w' \neq \varepsilon$ and $w' \neq w$, $\delta^*(i, w') \leq k$, i.e, if M starts in i and reads w, then it ends in j and during this computation, M only entered states from the set $\{1, \ldots, k\}$ (but the i in

the beginning and the j in the end do not count). We will start with $k = 0$ and then go on by induction.

Proof. By induction on k, we construct expressions $E^k_{i,j}$ such that

$$L(E^k_{i,j}) = \{w \mid \begin{array}{l}\delta^*(i,w) = j \text{ and for each prefix } w' \text{ of } w \\ \text{with } w' \neq \varepsilon \text{ and } w' \neq w,\ \delta^*(i,w') \leq k\}\end{array} \tag{3.1}$$

for all $1 \leq i, j \leq n$ and $0 \leq k \leq n$.
Induction base: If $k = 0$, then M is not allowed to enter any intermediate state when going from i to j. If $i \neq j$, then this means that M can only take an arc directly from i to j. Let $\sigma_1, \ldots, \sigma_t$ be the labels of the arcs from i to j. Then $E^0_{i,j} = \sigma_1 + \sigma_2 + \cdots + \sigma_t$ with the convention that this means $E^0_{i,j} = \emptyset$ if $t = 0$, i.e., there are no direct arcs from i to j. If $i = j$, then let $\sigma_1, \ldots, \sigma_t$ be the labels of all arcs from i to itself. Now we set $E^0_{i,i} = \varepsilon + \sigma_1 + \cdots + \sigma_t$. It is clear from the construction that $E^0_{i,j}$ fulfills (3.1).
Induction step: Assume that we have found regular expressions such that (3.1) holds for some k. We have to construct the expressions $E^{k+1}_{i,j}$. A path that goes from i to j such that all states in between are from $\{1, \ldots, k+1\}$ can either go from i to j by only going through states from $\{1, \ldots, k\}$. For this, we know already a regular expression, namely $E^k_{i,k}$. Or, when going from i to j, we go through $k+1$ exactly once. But we can view this as going from i to $k+1$ and all intermediate states are from $\{1, \ldots, k\}$ and then going from $k+1$ to j and again, all intermediate states are from $\{1, \ldots, k\}$. For the first part, we have already designed a regular expression, namely $E^k_{i,k+1}$, but also for the second part we have one, namely $E^k_{k+1,j}$. Their concatenation, $E^k_{i,k+1}E^k_{k+1,j}$, describes all words such that the automaton started in i, goes only through states $\{1, \ldots, k+1\}$ and only once through $k+1$, and ends in j. In general, if we go from i to j and all intermediate states are from $\{1, \ldots, k+1\}$, then we go from i to $k+1$ and all intermediate states are from $\{1, \ldots, k\}$, after that we may go several times from $k+1$ to $k+1$ and all intermediate states are from $\{1, \ldots, k\}$ and finally we go from $k+1$ to j and all intermediate states are from $\{1, \ldots, k\}$. The corresponding expression is $E^k_{i,k+1}(E^k_{k+1,k+1})^*E^k_{k+1,j}$. Altogether, we have $E^{k+1}_{i,j} = E^k_{i,j} + E^k_{i,k+1}(E^k_{k+1,k+1})^*E^k_{k+1,j}$. It is clear that $L(E^{k+1}_{i,k})$ is a subset of the language on the right-hand side of (3.1). But also the converse is true: Let $x_1, \ldots, x_s$ be all prefixes of w (sorted by increasing length) such that $\delta^*(i, x_i) = k+1$, $1 \leq i \leq s$, and define y_i by $x_{i+1} = x_i y_i$. Then $\delta^*(i, x_1) = k+1$, $\delta^*(k+1, y_i) = k+1$ for $1 \leq i < s$, and $\delta^*(k+1, y_s) = j$ where y_s fulfills $x_s y_s = w$. But this means that $w \in L(E^{k+1}_{i,j})$. This proves (3.1).

From (3.1), it follows that $L(M) = L(E^n_{1,j_1} + \cdots + E^n_{1,j_\ell})$ where $Q_{\text{acc}} = \{j_1, \ldots, j_\ell\}$. ∎

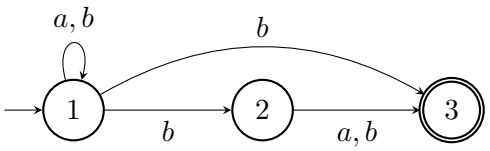
Remark 3.7 *The algorithm above does essentially the same as the* Floyd–Warshall *algorithm (see e.g. [CLRS09, Chapter 25]) for computing all-pair shortest paths. If we replace* $E_{i,j}^{k+1} = E_{i,j}^k + E_{i,k+1}^k (E_{k+1,k+1}^k)^* E_{k+1,j}^k$ *by* $d_{i,j}^k = \min\{d_{i,j}^k, d_{i,k+1}^k + d_{k+1,j}^k\}$*, we can compute the shortest distances between all pairs of nodes.*

3.4 Exercises

Basic exercises

Exercise 3.4 *Convert the following regular expression into an equivalent automaton:* $(a^\star b + bc)^\star$.

Exercise 3.5 *Convert the following deterministic finite automaton into an equivalent regular expression:*



Exercise 3.6 *Construct regular expressions for the following languages:*

1. $A = \{w \in \{0,1\}^* \mid w$ *contains an even number of* $0s\}$.
2. $B = \{w \in \{0,1\}^* \mid |w|$ *is divisible by* $3\}$.
3. $L_k = \{w \in \{0,1\}^* \mid$ *the* k*-last symbol is a* $1\}$

Intermediate exercises

Exercise 3.7 *Let* $k \in \mathbb{N}$ *be given. Construct a regular expression for the following language:*

$L_k = \{x \in \Sigma^\star \mid \exists i, j : 1 \leq i \leq i + k < j \leq j + k \leq |x|$ *und* $x_i \dots x_{i+k} = x_j \dots x_{j+k}\}$.

4 The pumping lemma

To show that a language L is regular, it is sufficient to give a deterministic finite automaton M with $L = L(M)$ or a nondeterministic one or a regular expression. But how do we show that a language L is not regular? Are there non-regular languages at all? Here is one example:

$$A = \{0^n 1^n \mid n \in \mathbb{N}\}.$$

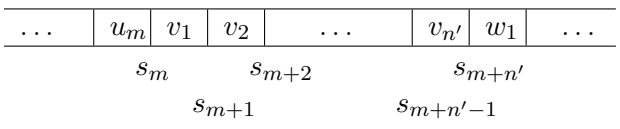
If there were a finite automaton for A, then it would have to keep track of the number of 0's that it read so far and compare it with the number of 1's. But with a finite number of states, you can only store a finite amount of information. But M potentially has to be able to store an arbitrarily large amount of information, namely n. (**Warning!** Never ever write this in an exam! This is just an intuition. Maybe there is a way other than counting to check whether the input is of the form $0^n 1^n$—there is not, but you have to give a formal proof.)

4.1 The pumping lemma

Lemma 4.1 (Pumping lemma) *Let L be a regular language. Then there is an $n > 0$ such that for all words u, v, w with $uvw \in L$ and $|v| \geq n$, there are words x, y, z with $v = xyz$ and $|y| > 0$ such that for all $i \in \mathbb{N}$, $uxy^izw \in L$.*

Proof overview: We choose n to be the number of states of a finite automaton M that recognizes L. Let $|v| = n' \geq n$. In the part of the computation of M on v, at least one state has to repeat by the pigeon hole principle. This means that we have discovered a loop in the computation of the automaton. By going through this loop i times instead of one time, we get the words uxy^izw.

Below, there is a automaton with uvw on the input tape. Below, there are the states of M when moving from one cell to the next (written in two lines because of the long indices). By the pigeon hole principle, two of them are equal, say, $s_{j_1} = s_{j_2}$ with $j_1 < j_2$. The word $v_{j_1+1} \dots v_{j_2}$ between them can be "pumped".



Proof. Since L is regular, it is accepted by a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$. Let $n = |Q|$. Let u, v, w be words such that $uvw \in L$ and $|v| \geq n$. Let $|u| = m$, $|v| = n' \geq n$, and $|w| = \ell$. Since $uvw \in L$, there is an accepting computation $s_0, s_1, \ldots, s_{m+n'+\ell}$ of M on uvw, i.e., $s_0 = q_0$, for all $0 \leq j \leq m + n' + \ell$

$$s_{j+1} = \begin{cases} \delta(s_j, u_{j+1}) & \text{if } 0 \leq j < m, \\ \delta(s_j, v_{j+1-m}) & \text{if } m \leq j < m + n', \\ \delta(s_j, w_{j+1-m-n'}) & \text{if } m + n' \leq j < m + n' + \ell, \end{cases}$$

and $s_{m+n'+\ell} \in Q_{\text{acc}}$. Since $s_m, \ldots, s_{m+n'}$ are more than n states, there are indices $m \leq j_1 < j_2 \leq m + n'$ such that $s_{j_1} = s_{j_2}$ by the pigeon principle. Let $v = xyz$ with $|x| = j_1 - m$ and $|y| = j_2 - j_1 > 0$. Then $\delta^*(q_0, ux) = s_{j_1}$, $\delta^*(s_{j_1}, y) = s_{j_2}$, and $\delta^*(s_{j_2}, zw) \in Q_{\text{acc}}$. Since $s_{j_1} = s_{j_2}$,

$$\begin{aligned} \delta^*(q_0, uxy^i zw) &= \delta^*(s_{j_1}, y^i zw) \\ &= \delta^*(s_{j_1}, y^{i-1} zw) \\ &\vdots \\ &= \delta^*(s_{j_1}, yzw) \\ &= \delta^*(s_{j_1}, zw) \\ &\in Q_{\text{acc}}. \end{aligned}$$

Thus $uxy^i zw \in L$ for all $i \in \mathbb{N}$. ∎

The statement "for all words u, v, w with $uvw \in L$ and $|v| \geq n$" in the pumping lemma can be rephrased as "for every decomposition of a word $s \in L$ into $s = uvw$ with $|v| \geq n$". This might or might not be more intuitive.

4.2 How to apply the pumping lemma

To show that a language L is not regular, one can show that L does not fulfill the condition of the pumping lemma since the contraposition of the statement of the pumping lemma says that if L does not fulfill the condition then it is not regular.

Example 4.2 *Let us show that $A = \{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular. The contraposition of the pumping lemma says that if for all n there are words u, v, w with $uvw \in A$ and $|v| \geq n$ such that for all words x, y, z with $xyz = v$ and $|y| > 0$, there is an $i \in \mathbb{N}$ such that $uxy^i zw \notin A$, then A is not regular. Let n be given. We set $u = \varepsilon$, $v = 0^n$, and $w = 1^n$. Obviously, $uvw \in A$ and $|v| \geq n$. Let $v = xyz$ with $|y| > 0$. Since $v = 0^n$, $x = 0^r$, $y = 0^s$, and $z = 0^t$ with $r + s + t = n$ and $s > 0$. We have $uxy^i zw = 0^{n+(i-1)s} 1^n$.*

Except for $i = 1$, this is not a word in A. (Even setting $i = 0$, i.e., "pumping down" gives a contradiction here.) Thus A does not fufill the condition of the pumping lemma and therefore, A is not regular.

We know that a finite language L is regular. Does not the pumping lemma imply that L contains an infinite number of words? The answers is of course no. The point is that the constant n from the pumping lemma will be larger than the length of any word in L. The the statement in the pumping lemma is automatically true, since there are no words in L with length at least n. In particular it follows from (the proof of) the pumping lemma that every deterministic finite automaton recognizing a finite language L has at least $\ell + 1$ states where ℓ is the maximum of length of a word in L.

Example 4.3 *Closure properties are sometimes useful to simplify proofs of non-regularity: Consider*

$$B = \{x \in \{0,1\}^* \mid \textit{the number of } 0\textit{'s equals the number of } 1\textit{'s in } x\}.$$

We have $A = B \cap L(0^*1^*)$. If B were regular, so would be A, a contradiction.

The pumping lemma game

Proving that a language L is not regular via the pumping lemma can be considered as a game between you and an adversary, for instance your professor.

1. Your professor picks an $n \in \mathbb{N} \setminus \{0\}$.
2. You pick words u, v, w such that $uvw \in L$ and $|v| \geq n$.
3. Your professor picks words x, y, z such that $v = xyz$ and $|y| > 0$.
4. You pick an $i \in \mathbb{N}$.

Now it comes to the showdown: You win if $uxy^izw \notin L$. Your professor wins if $uxy^izw \in L$. If you have a *winning strategy*, i.e., no matter what your professor picks, you can always make your choices such that you win, then L is not regular.

(If L is indeed not regular, this is one of the rare chances to win against your professor.)

The iron pumping lemma rule

The condition of the pumping lemma is only necessary.

To show that a language L is not regular you can show that it does not satisfy the condition of the pumping lemma. But you *cannot* prove that L is regular by showing that L fulfills the condition of the pumping lemma. There are non-regular languages that fulfill the condition of the pumping lemma.

The Schöning–Seidel–I don't know who version of the pumping lemma

In many textbooks or lecture notes of other lecturers you often find the following version of the pumping lemma:

Let L be a regular language. Then there is an $n > 0$ such that for all words t with $t \in L$ and $|t| \geq n$, there are words x, y, z with $t = xyz$, $|xy| \leq n$, and $|y| > 0$ such that for all $i \in \mathbb{N}$, $xy^iz \in L$.

Our version is more general than this version, since we can set $u = \epsilon$, v to the prefix of length n of t, and w to the rest of t.
Our version is much cooler, too. Let $L = \{0^i10^n1^n \mid i \geq 10, n \in \mathbb{N}\}$. In the version above, we cannot rule out that y is a substring of the first 10 zeroes and then pumping does not help. There are ways to work around this, but why do you want to make your life unneccesarily complicated. We can just choose $u = 0^i1$, $v = 0^n$, and $w = 1^n$ and are done.

4.3 Exercises

Basic exercises

Exercise 4.1 *Show that the following languages are not regular by using the pumping lemma:*

1. $A = \{0^n1^m \mid n > m\}$,
2. *B, the set of all palindromes over $\{0, 1\}$.*
   (A word is called a palindrome *if $x = x^{\text{rev}}$.)*

Exercise 4.2 *Show that no infinite subset of $\{0^n1^n \mid n \in \mathbb{N}\}$ is regular.*

Intermediate exercises

Exercise 4.3 *Show that the following languages are not regular by using the pumping lemma:*

1. $A = \{0^p \mid p \text{ is prime}\}$,
2. $B = \{0^{2^i} \mid i \in \mathbb{N}\}$.

Exercise 4.4 *We proved that* REG *is closed under (finite) intersection and union, that is, if $A, B \in$ REG so are $A \cap B$ and $A \cup B$. What about infinite intersection and union? More precisely, if $A_1, A_2, \cdots \in$ REG, are $\bigcap_{i=1}^{\infty} A_i$ and $\bigcup_{i=1}^{\infty} A_i$ regular?*

Exercise 4.5 *Which of the following languages are regular? Prove your claims!*

1. $A = \{0^n 1^{2m} \mid n, m \in \mathbb{N}\}$,
2. $B = \{0^n 1^m \mid n = 3m\}$,
3. $C = \{0^{p-1} \mid p \text{ is prime}\}$,
4. $D = \{0^n 1^m \mid n - m \leq 9001\}$,
5. $E = \{0^n 1^m \mid n \geq m \text{ and } m \leq 420\}$,
6. $F = \{0^n 1^m \mid n \geq m \text{ and } m \geq 420\}$,
7. $G = L((0^*1)^*0^*)$.

The language

$$\{0^n 1^m \mid n \neq m\}$$

is not regular, since we can write

$$\{0^n 1^m \mid n = m\} = (\{0,1\}^* \setminus \{0^n 1^m \mid n \neq m\}) \cap L(0^*1^*)$$

and REG is closed under complementation and intersection. Proving this directly via the pumping lemma turns out to be a little bit tricky.

Exercise 4.6 *Prove that $L = \{0^n 1^m \mid n \neq m\}$ is not regular using the pumping lemma directly.*

Advanced exercises

The next exercise generalizes the principle from Exercise 4.3.

Exercise 4.7 *Recall that a set $U \subseteq \mathbb{N}$ is called* ultimately periodic *if there are $n_0 \in \mathbb{N}$ and $p > 0$ such that for all $n \geq n_0$, $n \in U$ iff $n + p \in U$. p is called the* period *of U.*
Show the following: $L \subseteq \{0\}^*$ is regular iff $E = \{e \in \mathbb{N} \mid 0^e \in L\}$ is ultimately periodic.

The pumping lemma proven in the this chapter can only be used to prove that a language is not regular. We cannot use it to prove that a language is regular. In the next exercise, we construct an example. In Exercise 5.7, we prove a generalization of the pumping lemma that is neccessary and sufficient.

Exercise 4.8 *Let A be some language. Let*

$$L := \{x \in \{0,1\}^* \mid x \in A \text{ or } x \text{ contains } 00 \text{ or } 11 \text{ as a subword}\}$$

Prove the following:

1. *There is an $n \in \mathbb{N}$, such that for all u, v, w with $uvw \in L$ and $|v| \geq n$, there are words x, y, z with $v = xyz$ and $|y| > 0$ such that for all $i \in \mathbb{N}$, $uxy^izw \in L$.*
2. *We can choose A in such way that L is not regular.*

B Interlude: More on relations

B.1 Equivalence relations

Another important type of relations are *equivalence relations*. These are relations that are reflexive, symmetric and transitive.

Example B.1 *$E = \{(0,0),(1,1),(2,2)\}$ is the equality relation on $\{0,1,2\}$.*

E is an equivalence relation. It is easy to check that it is reflexive and it is trivially transitive and symmetric since there are no $(a,b) \in E$ with $a \neq b$. The relation E is the equality relation on the set $\{0,1,2\}$.

Example B.2 *Since $\{0,1,2\} \subseteq \{0,1,2,3\}$, we can consider E as a relation on $\{0,1,2,3\}$, too. Now E is not an equivalence relation any more, since $(3,3) \notin E$, that is, E is not reflexive.*

If R is an equivalence relation on a set A, then the *equivalence class* $[a]_R$ of an element $a \in A$ is $\{b \in A \mid (a,b) \in R\}$. If R is clear from the context, we will often write $[a]$ for short. Every element in an equivalence class is called a *representative* of this class.

Example B.3

1. *The relation E has three equivalence classes, namely $\{0\}$, $\{1\}$, $\{2\}$. Every element forms its own equivalence class.*
2. *Let $m \in \mathbb{Z}$, $m > 1$. Consider the relation M_m on $\mathbb{Z}$ given by*

$$(x,y) \in M_m :\iff m|x-y.$$

This is an equivalence relation: M_m is reflexive, since $m|0$. It is symmetric, since m divides a iff m divides $-a$. It is transitive, too: If m divides $x-y$ as well as $y-z$, then m also divides $x-y+y-z = x-z$. There are m equivalence classes, namely,

$$\{x \in \mathbb{Z} \mid x \equiv r \bmod m\}, \quad \text{for } r = 0,\ldots,m-1.$$

Lemma B.4 *Let R be an equivalence relation on a set A. Then for all $a,b \in A$, aRb iff $[a]_R = [b]_R$.*

Proof. "⇒": Let $x \in [a]_R$, that is, $(x,a) \in R$. Since $(a,b) \in R$, we have $(x,b) \in R$ by transitivity. Therefore, $x \in [b]_R$. Since x was arbitrary, $[a]_R \subseteq [b]_R$. $[b]_R \subseteq [a]_R$ follows in the same way. "⇐": Since R is reflexive, $b \in [b]_R$. Since $[a]_R = [b]_R$, we also have $b \in [a]_R$. Thus $(b,a) \in R$ (and $(a,b) \in R$ by symmetry). ∎

Exercise B.1 *Prove the following: The equivalence classes of an equivalence relation R on A form a* partition *of A, i.e., every $a \in A$ belongs to exactly one equivalence class of R.*

The *index* of an equivalence relation R is the number of equivalence classes of R and is denoted by $\text{index}(R)$. (If the number of equivalence classes is not finite, then $\text{index}(R)$ is infinite.)

We call an equivalence relation S on a set A a *refinement* of another equivalence relation R on A, if for every equivalence class C of S, there is an equivalence class D of R such that $C \subseteq D$.

Exercise B.2 *Let S and R be equivalence relations on some set A with finite index. Prove that if S is an refinement of R and $\text{index}(S) = \text{index}(R)$, then $S = R$.*

B.2 Derived equivalence relations

We discuss two ways to create new equivalence relations out of given ones.

Lemma B.5 *Let U and V be sets and let R be an equivalence relation on U and let f be a total function $V \to U$. The relation S on V, defined by*

$$S = \{(x, y) \mid (f(x), f(y)) \in R\}$$

is an equivalence relation, too.

Proof. S is reflexive: $(x, x) \in S$ because $(f(x), f(x)) \in R$ by the reflexivity of R.

S is symmetric: If $(x, y) \in S$, then $(f(x), f(y)) \in R$. By symmetry of R, $(f(y), f(x)) \in R$. And therefore, $(y, x) \in S$.

S is transitive: If $(x, y), (y, z) \in S$, then $(f(x), f(y)), (f(y), f(z)) \in R$. By transitivity of R, $(f(x), f(z)) \in R$, and thus, $(x, z) \in S$. ∎

We denote the relation S by $f^{-1}(R)$.

Lemma B.6 *Let $R_1, R_2, R_3, \ldots$ be equivalence relations on a set U. Then $\bigcap_{i \in \mathbb{N}} R_i$ is an equivalence relation, too.*

Proof. Let $S := \bigcap_{i \in \mathbb{N}} R_i$.

S is reflexive: $(x, x) \in S$, since $(x, x) \in R_i$ for all $i \in \mathbb{N}$ by the reflexivity of R_i.

S is symmetric: If $(x, y) \in S$, then $(x, y) \in R_i$ for all $i \in \mathbb{N}$. Since each R_i is symmetric, $(y, x) \in R_i$ for all i. Thus, $(y, x) \in S$.

S is transitive: If $(x, y), (y, z) \in S$, then $(x, y), (y, z) \in R_i$ for all $i \in \mathbb{N}$. Since each R_i is transitive, $(x, z) \in R_i$ for all i. Hence, $(x, z) \in S$. ∎

Remark B.7 *In particular, the intersection of a finite number of equivalence relations is an equivalence relation, too.*

Exercise B.3 *Give an example of two equivalence relations such that their union is* not *an equivalence relation.*

B.3 Exercises

Basic exercises

Exercise B.4 *Let $f : \mathbb{Z} \to \{0, 1, 2\}$ be the mapping $n \mapsto n \bmod 3$. Consider the equivalence relation E from Example B.1. Determine the relation $f^{-1}(E)$.*

Exercise B.5 *What are the equivalence classes of $M_2 \cap M_3$ (see Example B.3)?*

Intermediate exercises

Exercise B.6 *Let U and V be sets and let R be an equivalence relation on U and let f be a total function $V \to U$. Prove that the equivalence classes of $f^{-1}(R)$ are of the form $f^{-1}(C)$ where C is an equivalence class of R.*

Exercise B.7 *Let $R_1, R_2, R_3, \ldots$ be equivalence relations on a set U. Then the equivalence classes of $\bigcap_{i\in\mathbb{N}} R_i$ are of the form $\bigcap_{i\in\mathbb{N}} C_i$, where C_i is an equivalence class of R_i for all i.*

5 The Myhill-Nerode theorem and minimal automata

If a language is regular, then it is recognized by a deterministic finite automaton, which has a finite number of states. Since every subset of $\mathbb{N}$ has a minimum, there is an automaton with a minimum number of states. Is there a systematic way to find this minimum number?

Recall that an equivalence relation is a relation that is reflexive, symmetric, and transitive. We here consider equivalence relations on Σ^*. Let R be such an equivalence relation. For $x \in \Sigma^*$, $[x]_R$ denotes the equivalence class of x, i.e., the set of all $y \in \Sigma^*$ such that xRy. The equivalence classes of R form a partition of Σ^*, that means, they are pairwise disjoint and their union is Σ^*. We call a relation *right invariant* (with respect to concatenation) if for all $x, y \in \Sigma^*$,

$$xRy \quad \Longrightarrow \quad xzRyz \text{ for all } z \in \Sigma^*.$$

Definition 5.1 (Automaton relation) *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a deterministic finite automaton such that δ is total. The relation $\equiv_M$ is defined on Σ^* by*

$$x \equiv_M y :\Longleftrightarrow \delta^*(q_0, x) = \delta^*(q_0, y).$$

Lemma 5.2 *For every deterministic finite automaton M, $\equiv_M$ is an equivalence relation that is right invariant and has a finite index.*

Proof. $\equiv_M$ is an equivalence relation by Lemma B.5, because $=$ on the set Q is an equivalence relation.

It is right invariant, because $\delta^*(q_0, xz) = \delta^*(\delta^*(q_0, x), z)$. If $x \equiv_M y$, then $\delta^*(q_0, x) = \delta^*(q_0, y)$ and

$$\delta^*(q_0, xz) = \delta^*(\delta^*(q_0, x), z) = \delta^*(\delta^*(q_0, y), z) = \delta^*(q_0, yz)$$

and therefore, $xz \equiv_M yz$.

Finally, the index of $\equiv_M$ is bounded from above by $|Q|$, thus it is finite. ∎

Remark 5.3 *If all states of M are reachable from the start state, i.e., for all $q \in Q$, there is an $x \in \Sigma^*$ such that $\delta^*(q_0, x) = q$, then* $\text{index}(\equiv_M) = |Q|$.

Let $L \subseteq \Sigma^*$, and let M be a deterministic finite automaton for it. $\equiv_M$ defines a relation on Σ^* that of course depends on M. If we take two different deterministic finite automata M_1 and M_2 for L, then the relations might be different. Next, we define a relation $\sim_L$ on Σ^* that is independent of the chosen automaton. As we will see, every other relation $\equiv_M$ will be a refinement of it, i.e, every equivalence class of $\equiv_M$ is contained in a class of $\sim_L$. The relation $\sim_L$ is even defined for languages that are not regular.

Definition 5.4 (Myhill-Nerode relation) *Let $L \subseteq \Sigma^*$. The* Myhill-Nerode relation $\sim_L$ *is defined on Σ^* by*

$$x \sim_L y :\iff [\text{for all } z \in \Sigma^*\text{: } xz \in L \iff yz \in L].$$

Example 5.5 *Consider the language $L = L(0^*10^*10^*)$. L is the language of all words in $\{0,1\}^*$ that contain exactly two 1's. We claim that $\sim_L$ has four equivalence classes:*

$$A_i = \{x \in \{0,1\}^* \mid \text{the number of 1's in } x \text{ equals } i\} \qquad \text{for } i = 0,1,2$$
$$A_{>2} = \{x \in \{0,1\}^* \mid \text{the number of 1's in } x \text{ is } > 2\}$$

If the number of 1's in a word x equals $i \leq 2$, then $xz \in L$ iff the number of 1's in z equals $2-i$. If a word x has more than two 1's, then $xz \notin L$ for any $z \in \{0,1\}^*$. Thus A_0, A_1 ,A_2, and $A_{>2}$ are indeed the equivalence classes of L. 0, 1, 11, and 111 are representatives of these classes.

Lemma 5.6 *For every $L \subseteq \Sigma^*$, $\sim_L$ is an equivalence relation that is right invariant.*

Proof. $\sim_L$ is an equivalence relation: For each fixed $z \in \Sigma^*$, the relation R_z defined by $(x,y) \in R_z$ iff $xz \in L \iff yz \in L$ is an equivalence relation by Lemma B.5. The Myhill-Nerode relation is the intersection of all R_z and therefore an equivalence relation by Lemma B.6.

To see that it is right invariant, let $x \sim_L y$. We have to show that $xw \sim_L yw$ for all $w \in \Sigma^*$. $xw \sim_L yw$ means that for all $z \in \Sigma^*$, $xwz \in L \iff ywz \in L$. But this is clear since $x \sim_L y$ means that for all $z' \in \Sigma^*$, $xz' \in L \iff yz' \in L$, in particular for $z' = wz$. ∎

5.1 The Myhill-Nerode theorem

Theorem 5.7 (Myhill-Nerode) *Let $L \subseteq \Sigma^*$. The following three statements are equivalent:*

1. *L is regular.*
2. *L is the union of some equivalence classes of a right invariant equivalence relation with finite index.*

3. $\sim_L$ has finite index.

Proof. "1. $\Longrightarrow$ 2.": If L is regular, then there is a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ with $L(M) = L$. The relation $\equiv_M$ is an equivalence relation that is right invariant. Its index is $\leq |Q|$ and hence finite. By definition,

$$\begin{aligned} L &= \{x \in \Sigma^* \mid \delta^*(q_0, x) \in Q_{\text{acc}}\} \\ &= \bigcup_{q \in Q_{\text{acc}}} \{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}. \end{aligned}$$

By the definition of the automaton relation, each set $\{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}$ is an equivalence class of $\equiv_M$.
"2. $\Longrightarrow$ 3.": Let R be a right invariant equivalence relation with finite index such that $L = [x_1]_R \cup \cdots \cup [x_t]_R$. We show that R is a refinement of $\sim_L$, that is, every equivalence class C of R is a subset of some equivalence class C' of $\sim_L$. This implies $\text{index}(\sim_L) \leq \text{index}(R)$. Since $\text{index}(R)$ is finite, $\text{index}(\sim_L)$ is finite, too.

Let $x, y \in \Sigma^*$ with xRy. If we can show that $x \sim_L y$, then we are done since this means that any two elements from an equivalence class of R are in the same equivalence class of $\sim_L$. Hence every equivalence class of R is contained in an equivalence class of $\sim_L$. Since R is right invariant,

$$xzRyz \qquad \text{for all } z \in \Sigma^*. \tag{5.1}$$

Since $L = [x_1]_R \cup \cdots \cup [x_t]_R$, R-equivalent words are either both in L or both not in L. Hence (5.1) implies

$$xz \in L \iff yz \in L \qquad \text{for all } z \in \Sigma^*.$$

Thus $x \sim_L y$.
"3. $\Longrightarrow$ 1.": Given $\sim_L$, we construct a deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ with $L = L(M)$. We set

The Myhill-Nerode automaton

- Q = the set of equivalence classes of $\sim_L$,
- $\delta([x]_{\sim_L}, \sigma) = [x\sigma]_{\sim_L}$ for all $\sigma \in \Sigma$,
- $q_0 = [\varepsilon]_{\sim_L}$,
- $Q_{\text{acc}} = \{[x]_{\sim_L} \mid x \in L\}$.

δ is defined in terms of representatives, thus we have to check that it is well-defined, that means, if $x \sim_L y$, then $x\sigma \sim_L y\sigma$. But this follows immediately from the right invariance of $\sim_L$.

It remains to verify that $L(M) = L$. An easy proof by induction shows that $\delta^*([\varepsilon]_{\sim_L}, x) = [x]_{\sim_L}$. Thus,

$$L(M) = \{x \mid \delta^*([\varepsilon]_{\sim_L}, x) \in Q_{\text{acc}}\} = \{x \mid [x]_{\sim_L} \in Q_{\text{acc}}\}.$$

By definition, $[x]_{\sim_L} \in Q_{\text{acc}}$ iff $x \in L$. Thus $L(M) = L$. ∎

The Myhill-Nerode theorem was independently proved by Myhill [Myh57] and Nerode [Ner58]. Fun fact: The pumping lemma was not proved by Pumping, but by Bar-Hillel, Perles, and Shamir [BHPS61].

Exercise 5.1 *Show that $\delta^*([\varepsilon]_{\sim_L}, x) = [x]_{\sim_L}$ for all $x \in \Sigma^*$ in the "3. $\Longrightarrow$ 1."-part of the proof of the Myhill–Nerode theorem.*

Example 5.8 *Let $A = \{0^n 1^n \mid n \in \mathbb{N}\}$. We have $0^i \not\sim_A 0^j$ for $i \neq j$, since $0^i 1^i \in A$ but $0^j 1^i \notin A$. Thus $[0^i]_{\sim_A}$ for $i \in \mathbb{N}$ are pairwise distinct equivalence classes. Thus the index of $\sim_A$ is infinite and A is not regular.*

Myhill-Nerode theorem versus Pumping lemma

Both results are tools to show that a language is not regular.

Pumping lemma: often easy to apply but does not always work

Myhill-Nerode theorem: always works but often it is quite some work to determine the equivalence classes of $\sim_L$. Keep in mind that to show that $\sim_L$ has infinite index it is sufficient to find an infinite number of equivalence classes—we do not have to find all of them.

5.2 The minimal automaton

Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ and $M' = (Q', \Sigma, \delta, q_0', Q_{\text{acc}}')$ be deterministic finite automata such that δ and δ' are total. M and M' are *isomorphic* if there is a bijection $b : Q \to Q'$ sucht that

1. for all $q \in Q$ and $\sigma \in \Sigma$, $b(\delta(q, \sigma)) = \delta'(b(q), \sigma)$,
2. $b(q_0) = q_0'$,
3. $b(Q_{\text{acc}}) = Q_{\text{acc}}'$.

Such a mapping b is called an *isomorphism*. The first condition says that the following diagram commutes:

$$\begin{array}{ccc} Q \times \Sigma & \xrightarrow{\delta} & Q \\ {\scriptstyle b}\downarrow & \downarrow{\scriptstyle \text{id}} & \downarrow{\scriptstyle b} \\ Q' \times \Sigma & \xrightarrow{\delta'} & Q' \end{array}$$

Together with the second and third condition, this means that two isomorphic automata are the same up to renaming the states. Being isomorphic is an equivalence relation. The definition of isomorphic seems to be "asymmetric", the bijection is from Q to Q'. However, we can take the inverse map $b^{-1} : Q' \to Q$ and apply it to the equations in the items 1 to 3 in the definition of isomorphic (and replace q in item 1 by $b^{-1}(q')$ where $q' = b(q)$). Thus the situation is indeed symmetric.

Theorem 5.9 *Let $L \subseteq \Sigma^*$ be a regular language.*

1. *Any deterministic finite automaton $M' = (Q', \Sigma, \delta', q_0', Q'_{\text{acc}})$ such that δ' is a total function and $L(M') = L$ has at least* $\text{index}(\sim_L)$ *states.*

2. *Every deterministic finite automaton with total transition function and* $\text{index}(\sim_L)$ *many states that recognizes L is isomorpic to the automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ constructed in the "3. $\Longrightarrow$ 1."-part of the proof of the Myhill–Nerode theorem.*

Proof. Item 1: When we combine the "1. $\Longrightarrow$ 2."- and the "2. $\Longrightarrow$ 3."-part of the proof of the Myhill-Nerode theorem, we see that the relation $\equiv_{M'}$ is a refinement of $\sim_L$. Thus $|Q'| \geq \text{index}(\equiv_{M'}) \geq \text{index}(\sim_L)$.
Item 2: Assume now that $|Q'| = |Q|$. This means that $\text{index}(\sim_L) = \text{index}(\equiv_{M'})$. Since $\equiv_{M'}$ is a refinement of $\sim_L$, the equivalence classes of both relations are the same and hence both equivalence relations are the same. In particular, we can just simply write $[x]$ for the equivalence class of x in any of the two relations. Let

$$\begin{array}{rccll} b: & Q' & \to & Q & \\ & q' & \mapsto & [x] & \text{where } x \text{ is chosen such that } (\delta')^*(q_0', x) = q' \end{array}$$

Since $|Q'| = |Q|$, every state of M' is reachable. Thus b is defined for every $q' \in Q'$. Second, we have to check that b is well-defined: If y fulfills $(\delta')^*(q_0', y) = q'$, too, then $x \equiv_{M'} y$ and hence $x \sim_L y$. Since $\equiv_{M'}$ is a refinement of $\sim_L$, b is certainly surjective, and because $|Q'| = |Q|$, it is a bijection, too.

To show that M' is the same automaton as M (up to renaming of the states), we have to show that

1. $\delta(q, \sigma) = b(\delta'(b^{-1}(q), \sigma))$ for all $q \in Q$, $\sigma \in \Sigma$,
2. $b(q_0') = [\varepsilon]$, and
3. $b(Q'_{\text{acc}}) = Q_{\text{acc}}$.

For the first statement, let $q = [x]$, and let $b^{-1}(q) = q'$. Then $b(\delta'(q', \sigma)) = [x\sigma]$ by the definition of b. The second statement follows from the fact that $(\delta')^*(q_0', \varepsilon) = q_0'$ and the definition of b. For the third statement, let

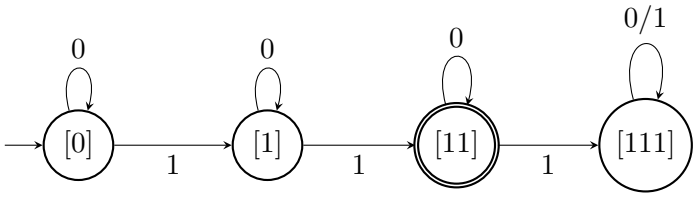


Figure 5.1: The minimal automaton for $L(0^*10^*10^*)$. It has four states that count the number of 1's.

$q' \in Q'_{\text{acc}}$ and let x be such that $(\delta')^*(q'_0, x) = q'$. Then $b(q') = [x]$. We have $x \in L(M') = L$ and thus $[x] \in Q_{\text{acc}}$. This shows $b(Q'_{\text{acc}}) \subseteq Q_{\text{acc}}$. This argument can be reversed, and thus we are done. ∎

Example 5.10 *We determined the equivalence classes of the Myhill-Nerode relation of $L = L(0^*10^*10^*)$ in Example 5.5. The corresponding minimal automaton is shown in Figure 5.1.*

5.3 An algorithm for minimizing deterministic finite automata

How do we actually find the minimal automaton for some language L? Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be any finite deterministic automaton with $L = L(M)$. We can assume that every state is reachable from q_0. We know that $\equiv_M$ is a refinement of $\sim_L$. The equivalence classes of $\equiv_M$ correspond to the states of M. Thus we have to identify these groups of states that form one equivalence class of $\sim_L$. Let A and A' be equivalence classes of $\equiv_M$ and assume that q and q' are the corresponding states, i.e., $x \in A \iff \delta^*(q_0, x) = q$ and $x \in A' \iff \delta^*(q_0, x) = q'$. A and A' are subsets of the same equivalence class of $\sim_L$ iff for all $z \in \Sigma^*$, $\delta^*(q, z) \in Q_{\text{acc}}$ iff $\delta^*(q', z) \in Q_{\text{acc}}$. A *proof* or *witness* that two states are not equivalent is a z with $\delta^*(q, z) \in Q_{\text{acc}}$ and $\delta^*(q', z) \notin Q_{\text{acc}}$. If there is such a z, then there is such a z that is short.

Lemma 5.11 *If there is a z with $\delta^*(q, z) \in Q_{\text{acc}}$ and $\delta^*(q', z) \notin Q_{\text{acc}}$, then there is a word z' with $|z'| \leq \binom{|Q|}{2}$ such that $\delta^*(q, z') \in Q_{\text{acc}}$ and $\delta^*(q', z') \notin Q_{\text{acc}}$ or vice versa.*

Proof. Let $s_0, s_1, \ldots, s_t$ be the computation of M when it starts in $q = s_0$ and reads z. Let $s'_0, s'_1, \ldots, s'_t$ be the computation of M when it starts in $q' = s'_0$ and reads z. If $t \leq \binom{|Q|}{2}$, then we are done. Otherwise, there are indices i and j, $i < j$, such that $\{s_i, s'_i\} = \{s_j, s'_j\}$ since $\binom{|Q|}{2}$ is the number

of unordered pairs with elements from Q. If $s_i = s_j$ and $s'_i = s'_j$, then, as in the proof of the pumping lemma, we can shorten the two computations by leaving out the states $s_{i+1}, \ldots, s_j$ and $s'_{i+1}, \ldots, s'_j$. The corresponding word is $z' = z_1 \ldots z_i z_{j+1} \ldots z_t$. If $s_i = s'_j$ and $s'_i = s_j$, then $s_0, \ldots, s_i, s'_{j+1}, \ldots, s'_t$ and $s'_0, \ldots, s'_i, s_j, \ldots, s_t$ are the computations of M on z' when started in $q = s_0$ and $q' = s'_0$. But now $\delta^*(q, z') \notin Q_{\text{acc}}$ and $\delta^*(q', z') \in Q_{\text{acc}}$. If z' is still longer than $\binom{|Q|}{2}$, we can repeat the process. ∎

To decide whether two states are equivalent, we "just" have to check whether for all $z \in \Sigma^*$ with $|z| \leq \binom{|Q|}{2}$, $\delta^*(q, z) \in Q_{\text{acc}}$ iff $\delta^*(q', z) \in Q_{\text{acc}}$. If q and q' are equivalent, then we can remove one of them, say q, and all arcs in the transition diagram that come into q now point to q' instead (i.e., if $\delta(p, \sigma) = q$, then $\delta(p, \sigma) = q'$ in the new automaton). The new automaton has one state less, and we can go on until we do not find a pair of equivalent states.

But there is a much faster algorithm. In some sense, words of length 1 are sufficient. The idea is to partition the set of states Q, that is, we write $Q = P_1 \cup \cdots \cup P_t$ such that $P_1, \ldots, P_t$ are pairwise disjoint. We define $[P_i] = \bigcup_{q \in P_i} \{x \mid \delta^*(q_0, x) = q\}$. For a single state q, we also write $[q]$ instead of $[\{q\}]$. We want to maintain the following invariant:

> For every equivalence class C of the Myhill-Nerode relation $\sim_{L(M)}$, there is an i such that $C \subseteq [P_i]$.

This ensures that for $p \in P_i$ and $q \in P_j$, $i \neq j$, $[p]$ and $[q]$ are contained in different Myhill-Nerode classes. We will start with the partition $P_1 = Q_{\text{acc}}$ and $P_2 = Q \setminus Q_{\text{acc}}$. This fulfills the invariant about, since words in $[Q_{\text{acc}}]$ and $[Q \setminus Q_{\text{acc}}]$ cannot be equivalent, as witnessed by ε.

Lemma 5.12 *Let $P_1, \ldots, P_t$ be a partition of the states of M as above. Let $1 \leq i, j \leq t$, $\sigma \in \Sigma$, and $q, p \in P_i$.*

1. *If $\delta(p, \sigma) \in P_j$ but $\delta(q, \sigma) \notin P_j$, then $[p]$ and $[q]$ are subsets of different Myhill-Nerode classes.*

2. *Conversely, if $[p]$ and $[q]$ are subsets of different Myhill-Nerode classes, then there is are indices i' and j', $\sigma' \in \Sigma$, and $p', q' \in P_{i'}$ such that $\delta(p', \sigma) \in P_{j'}$ and $\delta(q', \sigma) \notin P_{j'}$.*

The first statement says that when we separate the states of P_i into $\{p \in P_i \mid \delta(p, \sigma) \in P_j\}$ and $\{q \in P_i \mid \delta(q, \sigma) \notin P_j\}$, we keep our invariant. The second statement tells us that whenever our partition is to coarse, that is, there is an i such that $[P_i]$ contains Myhill-Nerode inequivalent words, then there is an index i' such that the separation step as described in 1 works for $P_{i'}$.

Proof. Let $x \in [p]$ and $y \in [q]$. Then $x\sigma \in [P_j]$ but $x\sigma \notin [P_j]$. This means that $x\sigma$ and $y\sigma$ are not Myhill-Nerode equivalent. Then x and y are not Myhill-Nerode equivalent, too, by right-invariance.

For the second statement, since $[p]$ and $[q]$ are subsets of different Myhill-Nerode classes, there is a string z such that $\delta^*(p, z) \in Q_{\text{acc}}$ and $\delta^*(q, z) \notin Q_{\text{acc}}$. Let z' be the longest prefix of z such that $p' := \delta^*(p, z')$ and $q' := \delta^*(q, z')$ are in the same set from $P_1, \ldots, P_t$. z' is a proper prefix of z since states from Q_{acc} and $Q \setminus Q_{\text{acc}}$ cannot occur in the same set by construction. Let σ' be the next symbol in z after z'. Then $\delta(p', \sigma')$ and $\delta(q', \sigma')$ are in different sets from $P_1, \ldots, P_t$ by the maximality of z'. ∎

Now the algorithms works as follows: We start with the initial partition $P_1 = Q_{\text{acc}}$ and $P_2 = Q \setminus Q_{\text{acc}}$. $x \in [P_1]$ and $y \in [P_2]$ cannot be Myhill-Nerode equivalent (as witnessed by ε). So our invariant is fulfilled initially. In each step of the algorithm, we choose one of the sets P_j and a $\sigma \in \Sigma$. Let $X = \{q \in Q \mid \delta(q, \sigma) \in P_j\}$. We replace each other P_h by $P_h \cap X$ and $P_h \setminus X$. If one of these two sets is empty, then P_h is not changed. It follows from the first part of Lemma 5.12 that we keep our invariant. If there does not exists a choice of j and σ such that the current partition changes, then we have found the Myhill-Nerode classes. This follows from the second part of the lemma. Below you can find an "implementation" of the algorithm in pseudocode.

Algorithm Hopcroft's minimization algorithm

Input: A deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$
with total transition function δ

Output: A partition $P_1, \ldots, P_t$ of Q into equivalent states, that is,
$\{x \mid \delta^*(q_0, x) \in P_i\}$ is a Myhill-Nerode class for all $1 \leq i \leq t$.

1: Start with $P_1 := Q_{\text{acc}}$ and $P_2 := Q \setminus Q_{\text{acc}}$.
2: **while** there are an i, j and $p, q \in P_i$ and $\sigma \in \Sigma$ such that $\delta(p, \sigma) \in P_j$ and $\delta(q, \sigma) \notin P_j$ **do**
3: Choose such a j and σ.
4: Set $X := \{q \in Q \mid \delta(q, \sigma) \in P_j\}$.
5: Replace every P_h by $P_h \cap X$ and $P_h \setminus X$.
If one of these sets is empty, then P_h is not changed.
{At least P_i will change by the condition of the loop.}
6: **od**
7: Return the current partition.

This algorithm can be speed up by carefully choosing the value j. The resulting algorithm is known as Hopcroft's algorithm [Hop71] and has a running time of $O(|\Sigma| n \log n)$.

5.4 Exercises

Basic exercises

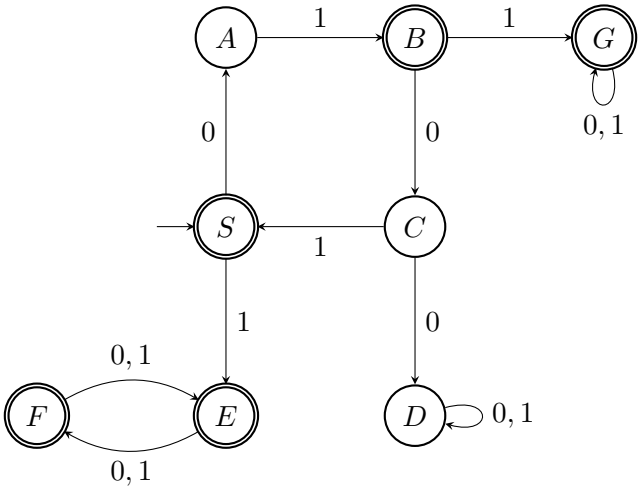
Exercise 5.2 *Determine the Myhill-Nerode classes of the following languages and construct the corresponding minimal automaton:*

1. $A = \{x \in \{0,1\}^* \mid x \text{ has an even number of } 0\}$,
2. $B = \{x \in \{0,1\}^* \mid x \text{ starts with a } 0 \text{ or ends with a } 1\}$.

Exercise 5.3 *For each of the following languages, give an infinite sequence of words that are pairwise Myhill-Nerode inequivalent.*

1. $A = \{0^n 1^m \mid n \geq m\}$,
2. $B = \{x \in \{0,1\}^* \mid x = x^{\text{rev}}\}$,
3. $C = \{0^{n^2} \mid n \in \mathbb{N}\}$.

Exercise 5.4 *Minimize the following automaton:*



First check whether the transition function is total.

Intermediate exercises

Exercise 5.5 *Let* $S_n = \{x \in \{0,1\}^* \mid \text{the } n\text{-last symbol of } x \text{ is a } 1\}$.

1. *Show that* $\text{index}(\sim_{S_n}) \geq 2^n$. *Conclude that the power set construction that transforms a nondeterministic finite automaton into a deterministic one is essentially optimal.*
2. *Give a regular expression* E_n *for* S_n *such that* $|E_n|$, *the length of* E_n *considered as a string, is linear.*

By the last exercise, every deterministic finite automation for the language S_n has an exponential number of states. However, we constructed a nondeterministic one with only $n+1$ states in Example 2.9 and a regular expression for S_n of linear size in the previous exercise.

The next exercise clarifies the relation between nondeterministic finite automata and regular expressions.

Exercise 5.6 *Analyse the constructions of the previous chapters to show:*

1. *Let $L = L(E)$ for a regular expression with $|E| = n$. (Here, $|E|$ is the length of E as a word.) Then there is a nondeterministic finite automation M for L with $O(n)$ many states.*

2. *Let $L = L(M)$ for a nondeterministic finite automaton M with n states. Then there is regular expression E for L of length $O(c^n n|\Sigma|)$.*

Advanced exercises

The next exercise is a generalization of the pumping lemma due to Jaffe [Jaf78] that is neccessary and sufficient.

Exercise 5.7 *A language $A \subseteq \Sigma^*$ has property (J), if*

there is an $n \geq 0$,
such that for all words $y \in \Sigma^*$ with $|y| = n$,
there are words u, v, w with $|v| > 0$ and $y = uvw$
and for all words z and $i \geq 0$: $yz \in A \iff uv^i wz \in A$.

1. *Prove that if A is regular, then it has property (J).*

2. *Prove that if A has property (J), then it is regular.*

There is another generalization by Stanat and Weiss [SW82].

Here is an alternative, really astonishing algorithm by Brzozowski [Brz63] for constructing the minimal deterministic automaton.

Exercise 5.8 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a finite deterministic automaton for some language L and assume that all states of M are reachable from q_0. Let M^{rev} be the automation that is obtained by reversing every edge in M and exchanging starting and accepting states. This automaton accepts the language L_{rev}, see Exercise 2.8. M^{rev} has multiple starting states, in the construction of Exercise 2.8 we added a new starting state with ε into these states. We do not do this here, but we keep the multiple starting states. M^{rev} accepts a string if there is an accepting computation starting in some starting state. Let N be the power set automaton of M^{rev}, see Theorem 2.8. The starting state of N will be Q_{acc}, which contains the staring states of M^{rev}.*

The accepting states are those set of states that contain q_0. In N we remove all states that are not reachable from Q_{acc}. Let $N = (P, \Sigma, \Delta, Q_{\text{acc}}, P_{\text{acc}})$ be the resulting automaton.

1. *Prove the following: For any states $q \in Q$ and $Z \in R$, we have $q \in \Delta^*(Z, w^{\text{rev}})$ iff $\delta^*(q, w) \in Z$.*
2. *Prove that for two states R, R' of N, if $[R]_{\equiv_N}$ and $[R]_{\equiv_N}$ are contained in the same Myhill-Nerode class, that is, $\Delta^*(R, x) \in P_{\text{acc}}$ iff $\Delta^*(R', x) \in P_{\text{acc}}$, then $R = R'$.*
3. *Conclude that N is the minimal automaton for L^{rev}.*
4. *Use this to build an algorithm for computing the minimum deterministic finite automaton*

The goal of the next exercise is to prove Theorem 2.12 (right after Exercise 2.11). There you can also find the definition of ultimately periodic and regularity preserving.

We consider a binary relation R on $\mathbb{N}$. For a language $L \subseteq \Sigma^*$, let

$$P(R, L) = \{x \mid \text{there is a } y \in \Sigma^* \text{ with } (|x|, |y|) \in R \text{ and } xy \in L\}.$$

We call R *regularity preserving* if for all regular language L, $P(R, L)$ is regular, too. R is called *u.p. preserving* if for any ultimately periodic set A,

$$R^{-1}(A) = \{m \mid \text{there is an } n \in A \text{ such that } (m, n) \in R\}$$

is ultimately periodic, too.

Exercise 5.9
1. *Prove that if R is regularity preserving, then R is u.p. preserving.*
2. *Prove that if R is u.p. preserving, then R is regularity preserving.*
3. *Exercise 2.11 is an immediate consequence of this characterization. What about other functions? For instance, prove that for every regular language L, the language*

$$L_{exp} = \{x \mid \textit{there is a } y \textit{ such that } |y| = 2^{|x|} \textit{ and } xy \in L\}$$

is regular.

I Twoway finite automata

I.1 Twoway finite automata

Would finite automata be more powerful if they could read the input more than once? Like an ordinary finite automaton a *deterministic twoway finite automaton* is described by a tuple $M = (Q, \Sigma, \vdash, \dashv, \delta, q_0, Q_{\text{acc}})$ such that

1. Q is a finite set, the set of states,
2. Σ is a finite set, the input alphabet,
3. $\vdash \notin \Sigma$ is the left end marker,
4. $\dashv \notin \Sigma$ is the right end marker,
5. $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{L, R\}$ is the transition function,
6. $q_0 \in Q$ is the start state,
7. $Q_{\text{acc}} \subseteq Q$ are the accepting states.

The only difference is that δ is now a function $Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{L, R\}$. If M is in state q, reads a symbol σ, and $\delta(q, \sigma) = (q', r)$, then the new state of M is q' and M moves its head to the left if $r = L$ and to the right otherwise. The symbols $\vdash$ and the $\dashv$ do not belong to the input alphabet; they mark the beginning and the end of the input word. Given an input w, we write $\vdash w \dashv$ on the input tape. The automaton starts on $\vdash$ and depending on the current state and the symbol it reads, changes its states and moves its head to the left or right. Whenever the automaton leaves the string $\vdash w \dashv$, it stops. It accepts the input w if it leaves the input to the right and is in an accepting state. We can define nondeterministic twoway finite automata, too. Here, δ maps into $\mathcal{P}(Q \times \{L, R\})$.

Example I.1 *Figure I.1 shows twoway deterministic automaton for the language* $S_3 = \{x \in \{0,1\}^* \mid$ *the third to last symbol is a* $0\}$. *This automaton can be extended to an automaton for* S_k *with* $k + 2$ *states. We know that every oneway deterministic finite automaton for* S_k *needs* 2^k *states by the Myhill-Nerode theorem.*

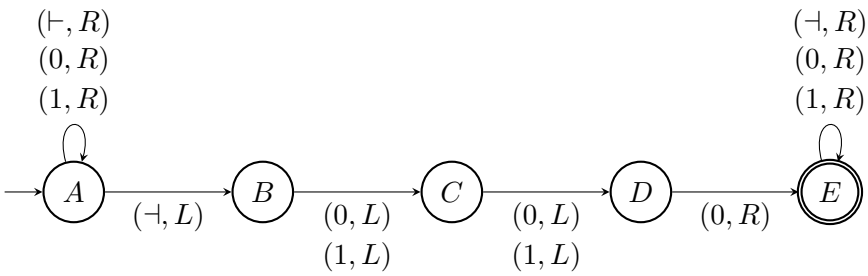


Figure I.1: A twoway deterministic finite automaton for S_3. In state A, the automaton goes to the right until it hits the right end marker $\dashv$. Then it goes three steps back and checks whether the third to last symbol is a 0. If yes, it leaves the input to the right in the accepting state E.

I.1.1 Computations

Computations for finite automata have been sequences of states so far. Finite automata also move their input head, but the position of the input head has been implicitly encoded in the position in a computation.

A *configuration* of a twoway finite automaton on input $x = x_1 \dots x_n$ is a tuple from $Q \times \{-1, \dots, n+2\}$. (q, i) describes the current state of the automaton and i the position on the input. $i = 0$ is the left end marker, $i = n+1$ is the right end marker. $i = -1$ means that the automaton left the input to the left and $i = n+2$ means that the input is left to the right.

We now define a binary relation $\vDash_x$ on the set of all configurations. Let $\vdash x_1 \dots x_n \dashv =: y_0 y_1 \dots, y_{n+1}$. Now, for all $p \in Q$ and $0 \leq i \leq n+1$,

$$\delta(p, y_i) = (q, L) \Rightarrow: (p, i) \vDash_x (q, i-1)$$
$$\delta(q, y_i) = (q, R) \Rightarrow: (p, i) \vDash_x (q, i+1)$$

The relation $\vDash_x$ describes one step of the automaton M. $(p, i) \vDash_x (q, i-1)$ means that if M is in state p and reads the symbol y_i, then it will enter state q and move to the symbol y_{i-1}.

If M is nondeterministic, then $\delta(p, y_i) = (q, L)$ is replaced by $(q, L) \in \delta(p, y_i)$. The same is done for the second line.

If $(p, i) \vDash_x (q, j)$, then we call (q, j) the *successor* configuration of (p, i). And (p, i) is called the *predecessor* of (q, j). The configuration $(q_0, 0)$ is called the *starting configuration* of M. Every configuration that does not have a successor is called a *halting configuration*. In particular, configurations of the form $(p, -1)$ and $(p, n+2)$ do not have any successor, that is, when M leaves the input, then the computation stops.

A *computation* of M on x is a sequence of configurations $C_1, \dots, C_t$ such that $C_\tau \vDash_x C_{\tau+1}$, $1 \leq \tau \leq t-1$, C_1 is the starting configuration, and C_t is a

halting configuration. In this case, M halts. A computation can also be an infinite sequence. In this case, M does not halt.

Now let $\vDash_x^*$ be the transitive closure of $\vDash_x$.

Definition I.2 *1. A twoway finite automaton M accepts a string x, if $(q_0, 0) \vDash_x^* (q, n+2)$ for some $q \in Q_{\text{acc}}$.*

2. $L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$.

This means that starting in $(q_0, 0)$, which is also called the starting configuration, the automaton leaves the input to the right in an accepting state. The definition works for deterministic and nondeterministic twoway finite automata, the the first case, there is only one possible computation, in the latter, there may be many. In the latter case, we can arrange the computations in a computation tree.

If an automaton does not accept a string x, then there are several possibilities: It stopped in a rejecting state, or it stopped in an accepting state but did not leave the input to the right, or it is looping in an infinite loop.

Example I.3 *Consider the automaton in Figure I.1. The computation on the string $x = 1010$ looks as follows:*

$$\begin{aligned}(A,0) &\vDash_x (A,1) \vDash_x (A,2) \vDash_x (A,3) \vDash_x (A,4) \vDash_x (A,5)\\ &\vDash_x (B,4) \vDash_x (C,3) \vDash_x (D,2)\\ &\vDash_x (E,3) \vDash_x (E,4) \vDash_x (E,5) \vDash_x (E,6)\end{aligned}$$

We end in an accepting configuration and leave the input to the right, therefore, we accept.

I.1.2 Crossing sequences

Let M be a twoway finite automaton (deterministic or nondeterministic). Fix an input x and fix a computation γ of M on x. A *crossing sequence* of M on input x at position i is the sequence of the states of M when moving its head from cell i to $i+1$ or from cell $i+1$ to i. We denote this sequence by $\text{CS}(x, i, \gamma)$. The odd positions in a crossing sequence correspond to moves of M from cell i to $i+1$ (left to right) and the even positions to moves from cell $i+1$ to i (right to left). (The cell 0 contains the $\vdash$ symbol, the cell $n+1$ the $\dashv$-symbol.) If $|x| = n$, then we have $n+1$ crossing sequences. We do not count the one step when M leaves the input to the right.

The crossing sequences of two neighbouring positions have to "match", i.e., if σ is the symbol in cell i and $(q_1, \ldots, q_k)$ and $(p_1, \ldots, p_\ell)$ are the crossing sequences at position i and $i+1$, then for instance, either $(p_1, R) \in \delta(q_1, \sigma)$ or $(q_2, L) \in \delta(q_1, \sigma)$, i.e., either M entered cell i in state q_1, read σ, changed its states to p_1 and moved to the right or it changed its state

to q_2 and moved to the left. To formalize this, we define two relations, a *left-matching* relation $\mathcal{M}_L$ and a *right-matching* relation $\mathcal{M}_R$. These two matching relations formalize what we outlined above: The left matching relation is for pairs of crossing sequences when M enters the cell i the first time by making a step to the left and the right matching relation is for the case that M enters the cell i the first time by a move to the right. In a computation of M, the former case cannot happen, since M always starts on the leftmost symbol. We need the left-matching relation for the inductive definition of the right-matching relation and vice versa.

Definition I.4 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a twoway finite automaton. We define the relations $\mathcal{M}_L$ and $\mathcal{M}_R$ inductively:*

1. *For all $\sigma \in \Sigma$, $((), \sigma, ()) \in \mathcal{M}_L$ and $((), \sigma, ()) \in \mathcal{M}_R$. Here $()$ denotes the empty sequence.*

2. *If $((q_3, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_R$ and $(q_2, L) \in \delta(q_1, \sigma)$, then $((q_1, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_R$.*

3. *If $((q_2, \ldots, q_k), \sigma, (p_2, \ldots, p_\ell) \in \mathcal{M}_L$ and $(p_1, R) \in \delta(q_1, \sigma)$, then $((q_1, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_R$.*

4. *If $((q_1, \ldots, q_k), \sigma, (p_3, \ldots, p_\ell) \in \mathcal{M}_L$ and $(p_2, R) \in \delta(p_1, \sigma)$, then $((q_1, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_L$.*

5. *If $((q_2, \ldots, q_k), \sigma, (p_2, \ldots, p_\ell) \in \mathcal{M}_R$ and $(q_1, L) \in \delta(p_1, \sigma)$, then $((q_1, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_L$.*

By construction of these relations, the claim of the following lemma is obvious.

Lemma I.5 *Let M be a twoway finite automaton and x be an input.*

1. *Let γ be an accepting computation on x and $S_0, \ldots, S_n$ be the crossing sequences at positions 0 to n. Then $(S_i, x_{i+1}, S_{i+1}) \in \mathcal{M}_R$ for all $0 \leq i < n$.*

2. *If we have crossing sequences $S_0, \ldots, S_n$ such that $(S_i, x_{i+1}, S_{i+1}) \in \mathcal{M}_R$ for all $0 \leq i < n$, then these crossing sequences constitute an accepting computation of M on x.*

I.1.3 Simulation

Theorem I.6 *The languages recognized by twoway finite automata are precisely the regular languages.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a twoway finite automaton. In principle, crossing sequences can be arbitrarily long. But if a crossing sequence has the same state in two odd or in two even positions, then the automaton performed a loop. It entered cell i in the same state and from the same direction. If M is deterministic, then this means that M is in an infinite loop. If M is nondeterministic, then M still might leave this loop because it can have other choices. But in this case, there is a shorter accepting computation. If M accepts an input x, then there is an accepting computation of M such that each crossing sequence in this computation has length at most $2|Q|$.

Now we define a oneway nondeterministic finite automaton $N = (Q', \Sigma, \delta', q'_0, Q'_{\text{acc}})$. that simulates M. The states of N are all crossing sequences of length at most $2|Q|$. The starting state of N is (q_0). The accepting states are those crossing sequences $(p_1, \ldots, p_\ell)$ of odd length such that $p_\ell \in Q_{\text{acc}}$. Finally,

$$\delta'((q_1, \ldots, q_k), \sigma) = \{(p_1, \ldots, p_\ell) \mid ((q_1, \ldots, q_k), \sigma, (p_1, \ldots, p_\ell) \in \mathcal{M}_R\}.$$

By Lemma I.5, M accepts x iff N accepts $\vdash x \dashv$.

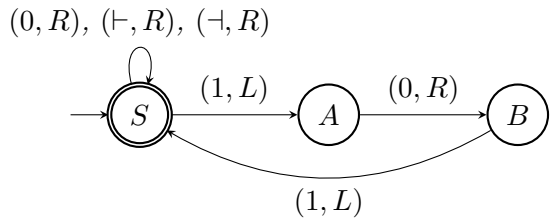
N accepts the language $\{\vdash x \dashv \mid x \in L(M)\}$. But it is very easy to get an automaton out of N that accepts $L(M)$. ∎

I.2 Exercises

Basic exercises

Exercise I.1 *Prove that a twoway finite automaton do not become more powerful when they have the option to stay on the current symbol, that is, δ is a function $Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{L, S, R\}$ (with the obvious semantics).*

Exercise I.2 *Consider the following twoway finite automaton:*



What language does it recognize? Can you convert it into a oneway finite automaton?

Research questions

We have seen that to simulate a oneway nondeterministic finite automaton by a oneway deterministic finite automaton, we might need exponentially

more states. To my best knowledge, it is still an open question whether there is an exponential gap when simulating twoway nondeterministic finite automata by a twoway deterministic finite automata, see [HS03] for some result in this direction.

Second, it is also not know whether there is an exponential gap when simulating oneway nondeterministic finite automata by twoway deterministic automata.

Part II

Computability

6 Limits of computations

In the first part, we have explored the limits of very simple systems, finite automata. Now we want to understand the limits of general computers. We will see that there are tasks that computers cannot do at all. Not just because you are missing some particular software or you are using the wrong operating system; we will reason about problems that a computer cannot solve no matter what.

One such task is verification: Our input is a computer program P and an input/output specification S.[1] S describes the desired input/output behaviour. We shall decide whether the program fulfills the specification or not. Can this task be automated? That means, is there a computer program V that given P and S returns 1, if P fulfills the specification and 0 otherwise? V should do this correctly for all possible pairs P and S.

Let us consider a (seemingly) easier task. Given a program P, decide whether P returns the value 0 on all inputs or not. That means, is there a program Z that given P as an input returns 1 if P returns the value 0 on all inputs and returns 0 if there is an input on which P does not return 0. (In constrast to finite automata, our programs compute with natural numbers instead of words over some finite alphabet. This is only done because it is more natural here, we will see soon that it does not matter at all.) How hard is this task? Does such a program Z exists? The following program indicates that this task is very hard (and one of the goals of the first part of this lecture is to show that it is impossible in this general setting).[2]

Program 1 expects four natural numbers as inputs that are initially stored in the variables $x_0, \ldots, x_3$. We do not specify its semantic formally at this point, but I am sure that you understand what Program 1 does. This program returns 1 on some input if and only if there are natural numbers $x_0 > 2$ and $x_1, x_2, x_3 \geq 1$ such that $x_1^{x_0} + x_2^{x_0} = x_3^{x_0}$. The famous Fermat's last theorem states that such four numbers do not exist. It took almost four hundred years until a valid proof of this conjecture was given.

Excursus: Fermat's last theorem

Pierre de Fermat (born 1607/08 in Beaumont de Lomagne, died 1665 in Castres, France) was a French lawyer who pursued mathematics as a "hobby". Nevertheless,

[1] This is just at an intuitive level. We will formalize this soon.

[2] Now it is a legitimate question why we should think about verification at all, when it is impossible. Well you should think about verification, *because* it is impossible. This is the real fun! While verification is impossible in general, it is still possible to solve interesting special cases or verify *one* particular program.

Program 1 Fermat

Input: $x_0, \ldots, x_3$
1: **if** $x_0 \leq 2$ **then**
2: return 0
3: **fi**
4: **if** $x_1 = 0$ or $x_2 = 0$ or $x_3 = 0$ **then**
5: return 0
6: **fi**
7: **if** $x_1^{x_0} + x_2^{x_0} = x_3^{x_0}$ **then**
8: return 1
9: **else**
10: return 0
11: **fi**

he is regarded as one of the greatest mathematicians. Fermat's last theorem states that the *Diophantine equation* $x^n + y^n = z^n$ does not have any integer solution for $n \geq 3$ and $x, y, z > 0$. The case $n = 1$ is of course trivial and for $n = 2$, the equation is fulfilled by all *Pythagorean triples*. Although it was always called Fermat's last theorem, it was an unproven conjecture written by Fermat in the margin of his copy of the Ancient Greek text *Arithmetica* by Diophantus. This note was discovered posthumously. The last part of this note became famous: "[...] cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet" (I have discovered a truly marvelous proof of this proposition. This margin is too narrow to contain it.) Fermat's last theorem was finally proven in 1995 by Andrew Wiles with the help of Richard Taylor.

Further reading: Simon Singh, *Fermat's last theorem*, Fourth Estate Ltd, 1997.

6.1 Exercises

Basic exercises

Exercise 6.1 Goldbach's conjecture *states that every even number* ≥ 4 *is the sum of two primes. Show that if you were able to write the program* Z*, then you could prove or disprove Goldbach's conjecture.*

7 WHILE and FOR programs

We want to prove mathematical statements about programs. Specifications of modern imperative programming languages like C++ or JAVA fill several hundred pages and do not typically specify everything. (You might remember some painful experiences.) As a first step, we will define a simple programming language called *WHILE* and specify its semantic completely in a mathematical model such that we can prove theorems about these programs. Then we will convince ourselves that C++ or JAVA programs cannot compute more than WHILE programs.

7.1 Syntax

Let us start with defining the syntax of WHILE programs. WHILE programs are strings over some alphabet. This alphabet contains

variables: x_0, x_1, x_2, ...

constants: 0, 1, 2, ...

key words: **while**, **do**, **od**

other symbols: $:=$, $\neq$, ;, $+$, $-$, [,]

Note that every variable is a symbol on its own and so is every constant. Also ":=" is treated as one symbol, we just write it like this in reminiscence of certain programming languages. (No programming language discrimination is intended.) Thus the underlying alphabet is infinite.

Definition 7.1 *WHILE programs are defined inductively:*

1. *A simple statement is of the form*

$$x_i := x_j + x_k \qquad \text{or}$$
$$x_i := x_j - x_k \qquad \text{or}$$
$$x_i := c,$$

   where $i, j, k \in \mathbb{N}$ and $c \in \mathbb{N}$.

2. *A WHILE program P is either a simple statement or of the form*

   (a) ***while*** $x_i \neq 0$ ***do*** P_1 ***od*** *or*

   (b) $[P_1; P_2]$

for some $i \in \mathbb{N}$ and WHILE programs P_1 and P_2.

We call the set of all WHILE programs $\mathcal{W}$. We give this set a little more structure. The set of all WHILE programs that consist of only one simple statement is called $\mathcal{W}_0$. We inductively define the sets $\mathcal{W}_n$ as follows:

$$\mathcal{W}_n = \mathcal{W}_{n-1} \cup \{P \mid \exists P_1 \in \mathcal{W}_{n-1},\, x_i \in X \text{ such that } P = \textbf{while } x_i \neq 0 \textbf{ do } P_1 \textbf{ od or}$$
$$\exists P_1 \in W_j,\, P_2 \in W_k \text{ with } j + k \leq n - 1 \text{ such that } P = [P_1; P_2]\}$$

Exercise 7.1 *Show the following: A WHILE program P is in $\mathcal{W}_n$ if and only if it was built by applying a rule from 2. in Definition 7.1 at most n times.*

Before we explain the semantics of WHILE programs, we first define another simple language, the *FOR language*. It uses the same elements as WHILE programs, we only have a different type of loop.

Definition 7.2 *FOR programs are defined inductively:*

1. *A simple statement is of the form*

$$x_i := x_j + x_k \qquad \text{or}$$
$$x_i := x_j - x_k \qquad \text{or}$$
$$x_i := c,$$

   where $i, j, k \in \mathbb{N}$ and $c \in \mathbb{N}$.

2. *A FOR program P is either a simple statement or it is of the form*

   (a) ***for*** x_i ***do*** P_1 ***od*** *or*

   (b) $[P_1; P_2]$

   for some $i \in \mathbb{N}$ and FOR programs P_1 and P_2.

The set of all FOR programs is denoted by $\mathcal{F}$. We define the subset $\mathcal{F}_n$ in the same manner as we did for $\mathcal{W}$. FOR programs differ from WHILE programs just by having a different type of loop.

7.2 Semantics

A program P gets a number of inputs $\alpha_0, \ldots, \alpha_{s-1} \in \mathbb{N}$. The input is stored in the variables $x_0, \ldots, x_{s-1}$. The output of P is the content of x_0 after the execution of the program. Note that in this way, the same WHILE or FOR program can have different numbers of inputs. Therefore, we have to specify in advance how many inputs the program is going to expect. This

will usually be clear from the context in what follows, therefore, we will often omit this specification.

The set $X = \{x_0, x_1, x_2, \dots\}$ of possible variables is infinite, but each WHILE or FOR program P always uses a finite number of variables. Let $\ell = \ell(P)$ denote the largest index of a variable in P. We always assume that $\ell \geq s - 1$. Intuitively, the behavior of a WHILE program P should only depend on the program itself and the content of the variables occuring in P. Therefore, we could model a memory state of P as a vector $S \in \mathbb{N}^{\ell+1}$. S_i would simply be the content of the variable x_i. (Some of the variables x_j with $j \leq \ell$ might not appear in P. But we do not care, in this case, S_j will remain unchanged during the whole execution of P.) While this looks fine, there is one small annoying inaccuracy. If $P = [P_1; P_2]$ and P_1 uses variables x_0, x_3, and x_7 and P_2 uses x_1, x_2, and x_9, then a state of P_1 is a vector of length 8 but states of P_2 and P have length 10. So a state of P_1 is formally not a state of P.

The "right" mathematical object to model the states of a WHILE program are sequences of natural numbers with *finite support*, i.e., functions $S : \mathbb{N} \to \mathbb{N}$ such that there is an $\ell_0 \in \mathbb{N}$ such that $S(n) = 0$ for all $n \geq \ell_0$. Therefore, a *state* of a WHILE or FOR program is a function $S : \mathbb{N} \to \mathbb{N}$ with finite support. Since only the values of S up to ℓ_0 are interesting, we will often just write down the values up to ℓ_0 and might even treat S as a finite vector in $\mathbb{N}^{\ell_0+1}$. The start state of a WHILE program P with s inputs on $\alpha_0, \dots, \alpha_{s-1}$ is the function $(\alpha_0, \dots, \alpha_{s-1}, 0, 0, \dots)$, which has finite support. (We can denote functions $\mathbb{N} \to \mathbb{N}$ by infinite vectors, which are simply the tables of values.)

Given a state S and a program $P \in \mathcal{W}_n$, we will now describe what happens when running P starting in S. This is done inductively, i.e., we assume that we already know how programs behave that are "smaller" than P where "smaller" means that they are built by less applications of the second rule in Definition 7.1 (or Definition 7.2), i.e., these programs are in $\mathcal{W}_{n-1}$. We now define a partial function Φ that defines the semantics of WHILE/FOR programs. It is a functions that maps states as defined above again to states. $\Phi_P(S)$ will denote the state that is reached after running P on state S. Φ_P will be a partial function, i.e., $\Phi_P(S)$ might be undefined (in the case when P does not halt on S). WHILE programs may not halt; at an intuitive level—we did not define the semantics so far—

```
1: x_1 := 1;
2: while x_1 ≠ 0 do
3:     x_1 := 1
4: od
```

is such a program.[1] The while loop never terminates and there is no state

[1]The assignment within the loop is necessary because the empty program is no valid WHILE program. There is no particular reason for this, we just defined it like this.

that can be reached since there is no "after the execution of the program".

1. If P is a simple statement then

$$\Phi_P(S) = \begin{cases} (\sigma_0, \dots, \sigma_{i-1}, \sigma_j + \sigma_k, \sigma_{i+1}, \dots) & \text{if } P \text{ is } x_i := x_j + x_k \\ (\sigma_0, \dots, \sigma_{i-1}, \max\{\sigma_j - \sigma_k, 0\}, \sigma_{i+1}, \dots) & \text{if } P \text{ is } x_i := x_j - x_k \\ (\sigma_0, \dots, \sigma_{i-1}, c, \sigma_{i+1}, \dots) & \text{if } P \text{ is } x_i := c \end{cases}$$

2. (a) Assume P equals **while** $x_i \neq 0$ **do** P_1 **od**. Let r be the smallest $r \in \mathbb{N}$ such that $\Phi_{P_1}^{(r)}(S)$ is either undefined, which means that P_1 does not terminate, or the ith position in $\Phi_{P_1}^{(r)}(S)$ equals 0.[2] Then

$$\Phi_P(S) = \begin{cases} \Phi_{P_1}^{(r)}(S) & \text{if } r \text{ exists and } \Phi_{P_1}^{(r)}(S) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

   (b) If P is $[P_1; P_2]$ for WHILE programs P_1 and P_2, then

$$\Phi_P(S) = \begin{cases} \Phi_{P_2}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S) \text{ and } \Phi_{P_2}(\Phi_{P_1}(S)) \text{ are both defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 7.3 *For every program P, Φ_P is well-defined.*[3]

Proof overview: The proof will be done by *structural induction*. Structural induction means that we assume that the statement is true for "simpler" programs and we show that the statement is valid for P, too. If $P \in \mathcal{W}_n$, then simpler will mean that the programs are in $\mathcal{W}_{n-1}$. In this way, structural induction becomes ordinary induction, since we now can just do induction on n.

Proof. Induction base: If $P \in \mathcal{W}_0$, then $\Phi_P(S)$ is defined for every S and it is only defined once; thus it is well-defined. This shows the induction basis.
Induction step: Let $P \in \mathcal{W}_n \setminus \mathcal{W}_{n-1}$ for $n > 0$. We assume that Φ_Q is well-defined for all programs $Q \in \mathcal{W}_{n-1}$. Since $n > 0$, P is either **while** $x_i \neq 0$ **do** P_1 **od** or $[P_1; P_2]$ for $P_1, P_2 \in \mathcal{W}_{n-1}$ and this decomposition is unique. By the induction hypothesis, Φ_{P_1} and Φ_{P_2} are well-defined. In both cases, the rules 2.(a) or 2.(b) explicitly define Φ_P and thus Φ_P is welldefined. ∎

[2] Here $f^{(r)}$ denotes the r-fold composition of functions, that is, $f^{(0)}(x) = x$ and $f^{(r)}(x) = f(f^{(r-1)}(x))$ for all x.

[3] What does well-defined mean? Since we did an inductive definition, we have to ensure that (1) each $\Phi_P(S)$ is defined or we explicitly say that the function value is undefined (since undefined means that P on S runs forever) and (2) we do not assign $\Phi_P(S)$ different values at different places.

The semantics of the simple statements is as we expect it: $x_i = x_j + x_k$ takes the values of x_j and x_k, adds them and stores the result in x_i. In case of subtraction, if the result is negative, we will set it to 0 instead, since we can only store values from $\mathbb{N}$. This operation is also called *modified difference*.

The syntax of a while loop is again as expected: We execute P_1 as long as the value of x_i does not equal 0. If the loop does not terminate, the result is undefined.

If P is the concatenation of P_1 and P_2, we first execute P_1 on S and then P_2 on the state produced by P_1 provided that P_1 halted.

In JAVA or other program languages, we do not have to put brackets around concatenations. We will now prove that we can use the same convention for WHILE programs, namely we show that concatenation is associative with respect to the function Φ:

Lemma 7.4 *For any three WHILE programs P_1, P_2, and P_3,*

$$\Phi_{[P_1;[P_2;P_3]]} = \Phi_{[[P_1;P_2];P_3]}.$$

Proof. By applying rule 2.(a), we have

$$\Phi_{[P_1;[P_2;P_3]]}(S) = \begin{cases} \Phi_{[P_2;P_3]}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S) \text{ and } \Phi_{[P_2;P_3]}(\Phi_{P_1}(S)) \text{ are both defined,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

and

$$\Phi_{[P_2;P_3]}(T) = \begin{cases} \Phi_{P_3}(\Phi_{P_2}(T)) & \text{if } \Phi_{P_2}(T) \text{ and } \Phi_{P_3}(\Phi_{P_2}(T)) \text{ are both defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

By setting $T = \Phi_{P_1}(S)$, if it is defined, we get that

$$\Phi_{[P_1;[P_2;P_3]]}(S) = \begin{cases} \Phi_{P_3}(\Phi_{P_2}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S), \Phi_{P_2}(\Phi_{P_1}(S)) \text{ and } \Phi_{P_3}(\Phi_{P_2}(\Phi_{P_1}(S))) \text{ are defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In the same way, we get that

$$\Phi_{[[P_1;P_2];P_3]}(S) = \begin{cases} \Phi_{P_3}(\Phi_{[P_1;P_2]}(S)) & \text{if } \Phi_{[P_1;P_2]}(S) \text{ and } \Phi_{P_3}(\Phi_{[P_1;P_2]}(S)) \text{ are both defined,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

and

$$\Phi_{[P_1;P_2]}(S) = \begin{cases} \Phi_{P_2}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S) \text{ and } \Phi_{P_2}(\Phi_{P_1}(S)) \text{ are both defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Thus, by plugging the second equation into the first,

$$\Phi_{[[P_1;P_2];P_3]}(S) = \begin{cases} \Phi_{P_3}(\Phi_{P_2}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S), \Phi_{P_2}(\Phi_{P_1}(S)) \text{ and } \Phi_{P_3}(\Phi_{P_2}(\Phi_{P_1}(S))) \text{ are defined,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

which finishes the proof. ■

Now we can simply write $P_1; P_2; P_3$ instead of $[[P_1; P_2]; P_3]$ or $[P_1; [P_2; P_3]]$. However, note that once we write $P_1; P_2; P_3$, we do not know any more which of the two programs was the original one. As long as we are only interested in semantics, that does not matter. However, syntactically, these two programs are different (they are not the same string!) and therefore, a syntactically correct WHILE program needs all the brackets $[\dots]$. We consider dropping these brackets as *syntactic sugar*.

The semantics of FOR programs is defined in the same manner as for WHILE programs. The semantics of the simple statements stays the same, we only have to modify the interpretation of for loops.

2. (a) Assume P equals **for** x_i **do** P_1 **od** for some FOR program P_1. Then
$$\Phi_P(S) = \Phi_{P_1}^{(\sigma_i)}(S).$$
(b) If P is $[P_1; P_2]$ for FOR programs P_1 and P_2, then
$$\Phi_P(S) = \Phi_{P_2}(\Phi_{P_1}(S)).$$

A for loop executes P_1 σ_i times, where σ_i is the value of x_i before the execution of the for loop. This means that changing the value of x_i during the execution of the for loop does not have any effect.[4] In particular, for loops always terminate.

This means that we can simplify the interpretation of the concatenation of FOR programs since we do not have to deal with undefined values of Φ. This makes is even easier to show that concatenation of FOR programs is associative with respect to Φ (c.f. Lemma 7.4).

Exercise 7.2 *Show that every FOR loop can be simulated by a WHILE loop. (Simulation here means that for every FOR program P of the form **for** x_i **do** P_1 **od** we can find a WHILE program Q such that $\Phi_P = \Phi_Q$, i.e., both programs compute the same function.)*

7.3 Computable functions and sets

Definition 7.5 *Let P be a WHILE or FOR program with s inputs. The function $\varphi_P^s : \mathbb{N}^s \to \mathbb{N}$ computed by P is defined by*

$$\varphi_P^s(\alpha_1, \dots, \alpha_s) = \begin{cases} \textit{first entry of } \Phi_P((\alpha_1, \dots, \alpha_s, 0, \dots)) \\ \qquad \textit{if } \Phi_P((\alpha_1, \dots, \alpha_s, 0, \dots)) \textit{ is defined} \\ \textit{undefined} \qquad \textit{otherwise} \end{cases}$$

for all $(\alpha_1, \dots, \alpha_s) \in \mathbb{N}^s$.

[4] For loops in C++, for instance, work differently; they are merely while loops.

If s is clear from the context, we will often simply write φ_P.

Definition 7.6 *1. A partial function $f : \mathbb{N}^s \to \mathbb{N}$ is* WHILE computable *if there is a WHILE program P such that $f = \varphi_P$.*

2. f is FOR computable *if there is a FOR program P such that $f = \varphi_P$.*

3. The class of all WHILE computable functions is denoted by R.

4. The class of all FOR computable functions is denoted by PR.

The acronyms R and PR stand for *recursive* and *primitive recursive*. These are the first names that were used for these two classes of functions and we also use them throughout the following.

By Exercise 7.2, $\mathsf{PR} \subseteq \mathsf{R}$. Since for loops always terminate, all functions in PR are total. On the other hand, there are functions that are not total but WHILE computable. For instance, we already saw a program that computes the function $\mathbb{N} \to \mathbb{N}$ that is defined nowhere. There are also total functions in $\mathsf{R} \setminus \mathsf{PR}$. We will show their existence later.

Most of the time, we will talk about subsets $L \subseteq \mathbb{N}$. We also call such sets *languages*, in analogy to the first part on automata. For any such L, we define its *characteristic function* to be the following function:

$$\chi_L : \mathbb{N} \to \{0,1\}$$
$$x \mapsto \begin{cases} 1 & \text{if } x \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Since $\{0,1\}$ is a subset of $\mathbb{N}$, we can view χ_L as a function $\mathbb{N} \to \mathbb{N}$.

Definition 7.7 *1. A language $L \subseteq \mathbb{N}$ is called* recursive *or* decidable *if $\chi_L \in \mathsf{R}$.*

2. The set of all recursive languages is denoted by REC.

7.4 Exercises

Basic exercises

Exercise 7.3 *Show that for each FOR program P, Φ_P is a total function.*

Hint: Use structural induction. The proof is close to trivial but/and a good exercise in structural induction.

Exercise 7.4 *1. Prove that the WHILE computable functions are closed under composition. More specifically, prove that when $f : \mathbb{N}^s \to \mathbb{N}$ and*

$g_1, \ldots, g_s : \mathbb{N}^t \to \mathbb{N}$ are WHILE computable, so is the function F given by

$$\begin{aligned} \mathbb{N}^t &\to \mathbb{N} \\ (x_1, \ldots, x_t) &\mapsto f(g_1(x_1, \ldots, x_t), \ldots, g_s(x_1, \ldots, x_t)). \end{aligned}$$

If one of the $g_i(x_1, \ldots, x_t)$ is undefined, then $F(x_1, \ldots, x_t)$ will be undefined, too.

2. *Conclude that WHILE computable functions $\mathbb{N} \to \mathbb{N}$ are closed under (ordinary) composition, addition and multiplication. (For multiplication you might want to read Chapter 8.4 first.)*

Remark 7.8 *The same closure properties hold for FOR computable functions by the very same proof.*

Intermediate exercises

Exercise 7.5 *Let P and P' be WHILE programs such that $\Phi_P = \Phi_{P'}$. Prove that*

1. *$\Phi_{P;Q} = \Phi_{P';Q}$ and $\Phi_{Q;P} = \Phi_{Q;P'}$ for all WHILE programs Q and*
2. *$\Phi_{\textbf{\textit{while}}\ x_i \neq 0\ \textbf{\textit{do}}\ P\ \textbf{\textit{od}}} = \Phi_{\textbf{\textit{while}}\ x_i \neq 0\ \textbf{\textit{do}}\ P'\ \textbf{\textit{od}}}$.*

Exercise 7.6 *Prove that* REC *is closed under union, intersection and complementation.*

We can restrict the set of simple statements even further. This will be useful in some proofs.

Exercise 7.7 *Prove that in WHILE or FOR programs, we can replace the simple statements by the statements $x_i{+}{+}$, $x_i{-}{-}$, and $x_i := 0$. Here the statement $x_i{+}{+}$ increases the value of x_i by 1 and $x_i{-}{-}$ decreases it by 1, unless the value of x_i is 0, then it is unchanged. (Formally, we now have to define the semantics of these modified programming languages by defining new functions Φ and φ. Then we have to prove that whenever a function is WHILE or FOR computable in the sense of Definiton 7.6, then it is computable by a WHILE or FOR program with the restricted set of simple statements.)*

Advanced exercises

The next exercise is a follow-up exercise to Exercise 7.7.

Exercise 7.8 *1. Prove that in the restricted FOR programs of Exercise 7.7, we can even drop the $x_i{-}{-}$ statements, that is, every FOR computable function can be computed by for loops and simple statements of the form $x_i{+}{+}$ and $x_i := 0$.*

2. Prove that there is a WHILE computable function that is not computable by while loops and simple statements of the form $x_i{+}{+}$ and $x_i := 0$. (Why can't you simply simulate for loops by while loops and use the first part?)

8 Syntactic sugar

The languages WHILE and FOR consist of five constructs. We now want to convince ourselves that we are still able to compute every function $\mathbb{N}^s \to \mathbb{N}$ that JAVA or C++ could compute. (And no, we are not able to display fancy graphics.)

8.1 Variable names

WHILE and FOR offers only boring variable names like x_{17}. But of course we can replace them by more fancy ones. We can always revert to ordinary WHILE/FOR programs by replacing the new fancy name by some x_i that has not been used so far.

8.2 Assignments

WHILE does not contain assignments of the form $x_i := x_j$. But this can be easily simulated by

1: $x_k := 0$;
2: $x_i := x_j + x_k$

where x_k is a new variable. (Since x_k is a new variable, it is initialized with 0. Therefore, we could even omit the first line.)

Remark 8.1 *Formally, we also have to prove the correctness of our simulation above. So for any state $S = (\sigma_0, \sigma_1, \dots)$ we have to prove that if $\Phi_P(S) = (\sigma'_0, \sigma'_1, \dots)$, then $\sigma'_i = \sigma_j$ and all other entries do not change except for k, where P is the program above. This is quite easy here. We have*

$$\Phi_{x_k:=0}((\dots,\sigma_i,\dots,\sigma_j\dots,\sigma_k,\dots)) = (\dots,\sigma_i,\dots,\sigma_j\dots,0,\dots) \qquad \textit{and}$$
$$\Phi_{x_i:=x_j+x_k}((\dots,\sigma_i,\dots,\sigma_j\dots,0,\dots)) = (\dots,\sigma_j,\dots,\sigma_j\dots,0,\dots),$$

where we have only shown the three relevant positions of the state and assumed $i \le j < k$ for simplicity.

Even more formally, you would even have to prove that when you have any WHILE program Q with assignments and you replace these assignments with the construction above yielding an ordinary WHILE program Q', then $\Phi_Q = \Phi_{Q'}$, where we define Φ_Q in the natural way. (Q is not a proper WHILE program!) Although the formal proof is a little tedious, it should be clear that this is doable. Exercise 7.4 is quite helpful.

8.3 Procedures and functions

Let h be a WHILE or FOR computable function $\mathbb{N}^t \to \mathbb{N}$, respectively, and let P be a WHILE or FOR program for h. We can enrich WHILE or FOR by statements of the form $x_i = h(x_{j_1}, \dots, x_{j_t})$ and can always revert back to ordinary WHILE or FOR as shown in Program 2.

Program 2 Simulation of a function call

1: $x_{\ell+1} := x_{j_1}$;
2: $\vdots$
3: $x_{\ell+t} := x_{j_t}$;
4: $x_{\ell+t+1} := 0$;
5: $\vdots$
6: $x_{\ell+m+1} := 0$;
7: $\hat{P}$;
8: $x_i := x_{\ell+1}$

In Program 2, ℓ is the largest index of a variable used in the current program and m is the largest index of a variable in P. $\hat{P}$ is the program obtained from P by replacing every variable x_i by $x_{i+\ell+1}$. Basically, we replace every occurence of h by a program that computes h and avoid any interferences by choosing new variables. This does not give a very short program but we do not care. (Note that this does not allow recursive function calls yet. We need to implement stacks to do this.)

Exercise 8.1 *The evaluation strategy above is* call by value. *How do you implement* call by reference*?*

8.4 Arithmetic operations

WHILE and FOR only offers addition and (modified) subtraction as arithmetic operations. But this is enough to do everything else. As an example, let us implement the multiplication $x_i := x_j \cdot x_k$:

1: $x_i := 0$;
2: **for** x_j **do**
3: $\quad x_i := x_i + x_k$
4: **od**

The implementation above assumes $i \neq j, k$. Since a FOR loop can be simulated by a WHILE loop, we can perform multiplications in WHILE, too.

Remark 8.2 *To formally prove the correctness of the construction above, we can show that for any state $S = (\sigma_0, \sigma_1, \dots)$ after executing the body of the for loop for the mth time, the ith component of the new state has value*

$m \cdot \sigma_k$ and all other entries are left unchanged. This can be easily done by induction.

Exercise 8.2 *Formalize the proof outlined in the remark above.*

Exercise 8.3 *Show that integer division does not increase the power of FOR programs, i.e., we can simulate computing the quotient $x_i := x_j/x_k$ and the remainder $x_i := x_j$ rem x_k in ordinary FOR.*

8.5 Stacks

Built-in types like `int` in C++ or JAVA can only store a limited amount of information. But we can get as many variables of this type as we want by using dynamic arrays (or `memalloc`, if you prefer that). WHILE programs only have a finite number of variables but each of them can store an arbitrarily large amount of information. In this way, we can simulate arrays. But before implementing arrays, we will implement stacks first.

Since built-in types like `int` only store a finite number of information, you will never be able to store the content of two `int`-variables in one `int`-variables. But infinite sets like natural numbers behave differently. We start with explaining how to "store" two natural numbers in one.

Lemma 8.3 *There are FOR computable functions $\langle ., . \rangle : \mathbb{N}^2 \to \mathbb{N}$ and $\pi_i : \mathbb{N} \to \mathbb{N}$, $i = 1, 2$, such that*

$$\pi_i(\langle x_1, x_2 \rangle) = x_i$$

for all $x_1, x_2 \in \mathbb{N}$ and $i = 1, 2$.

Proof. We define $\langle x_1, x_2 \rangle = \frac{1}{2}(x_1 + x_2)(x_1 + x_2 + 1) + x_2$. Note that either $x_1 + x_2$ or $x_1 + x_2 + 1$ is even, therefore, $\langle x_1, x_2 \rangle$ is a natural number. $\langle ., . \rangle$ is obviously FOR computable by the results in Section 8.4 and Exercise 8.3.

Let $p = \langle x_1, x_2 \rangle$ be given. We want to reconstruct x_1 and x_2 from p. We have

$$\frac{1}{2}(z + 1)(z + 2) - \frac{1}{2}z(z + 1) = z + 1.$$

Therefore, the largest z such that $\frac{1}{2}z(z+1) \leq p$ is $x_1 + x_2$, since $x_2 \leq x_1 + x_2$. From this sum $z = x_1 + x_2$, we can reconstruct x_2 as $p - \frac{1}{2}z(z + 1)$. Finally, from the z and x_2 we can reconstruct x_1 as $z - x_2$. In Exercise 8.4, we construct FOR programs for π_1 and π_2. ∎

Exercise 8.4 *Construct FOR programs for the projections π_i, $i = 1, 2$.*

Remark 8.4 *$\langle ., . \rangle$ is also surjective.*

While you can forget about the rest of the construction of stacks and arrays once you believe that we can simulate them, this pairing functions is essential for later chapters and you should not forget about its properties.

Excursus: Gauß's formula for consecutive integer sums

Carl Friedrich Gauß (1777–1855) was a German mathematician, astronomer, and geodesist. If he had lived today, he would also have been a computer scientist for sure.

One hears the following anecdote quite often:[1] At the age of seven, Gauß attended the Volksschule. To have some spare time, the teacher Büttner told the pupils to sum up the numbers from 1 to 100. Gauß quickly got the answer 5050 by recognizing that $1 + 100 = 101$, $2 + 99 = 101$, $\ldots 50 + 51 = 101$, in other words, $\sum_{i=1}^{100} i = \frac{1}{2} \cdot 100 \cdot 101$. Poor Büttner.

Gauß was also one of the main reasons against introducing the Euro.

A *stack* is a data structure that stores some objects, here our objects will be natural numbers. We can either *push* a number onto the stack. This operation stores the number in the stack. Or we can *pop* an element from the stack. This removes the element from the stack that was the last to be pushed onto the stack among all elements still in the stack. If the stack is empty and we want to pop an element from the stack, then the stack remains unchanged. So it works like a stack of plates where you can only either remove the top plate or put another plate on the top.[2] To access the stack, there is an operation *top*, which returns the element on top of the stack without changing it. If the stack is empty, then top can return any value. There is usually also a function *isempty* that allows you to check whether a stack is empty or not.

We represent a stack by $S = \langle n, Y \rangle$. (We are already using the convention from Section 8.1.) n is the number of elements in S and Y is the content of S. The empty stack is represents by $\langle 0, 0 \rangle$. If $a_1, \ldots, a_n$ are stored in S, with a_n being the element pushed onto S last, then $Y = \langle a_n, \langle a_{n-1}, \ldots \langle a_1, 0 \rangle \ldots \rangle \rangle$.

The push operation is implemented in Programm 3: In the first line we extract the current content of S by projecting onto the second component and then we push x onto the stack by using the pairing function. Then we increase the counter in line 2 and pair it with the new content.

Program 3 push(S, x)

1: $Y := \langle x, \pi_2(S) \rangle$;
2: $S := \langle \pi_1(S) + 1, Y \rangle$

---

[1] In this lecture notes, you occasionally find anecdotes about mathematicians and computer scientists. I cannot guarantee that they are true but they are entertaining. This particular one can be found on Wikipedia, which indicates that it is false.

[2] Dexterous people might do different things but computer scientist usually do not.

The pop operation is implemented in a similar fashion. First we extract the number of elements. If this number is nonzero, then we subtract 1 from it and remove the top element from the stack by using the projection π_2. Note that we used further syntactic sugar, namely, an if-then-statement, see Exercise 8.5 at the end of this chapter how to implement this.

Program 4 pop(S)

```
1: n := π1(S);
2: if n ≠ 0 then
3:     S := ⟨n − 1, π2(π2(S))⟩
4: fi
```

Next we show how to implement an operation of the form $x = \text{top}(S)$. This can simply be done by using our projection functions.

Program 5 $x = \text{top}(S)$

```
1: x := π1(π2(S))
```

Finally, isempty can be realized in Programm 6

Program 6 $b = \text{isempty}(S)$

```
1: n := π1(S);
2: if n > 0 then
3:     b := 0
4: else
5:     b := 1
6: fi
```

8.6 Arrays

Next, we implement arrays. We assume that an array A stores m elements $A[0], \ldots, A[m-1]$. We will use a stack to store the array, $A[0]$ will be the top element. For simplicity, we call the stack A again. In the beginning, A will contain only zeroes. Therefore, we can initialize A with 0.

To access $A[i]$, we simply pop i elements from the stack and then we return the top element. Of course, we should only do this with a copy of A, because the pop operation destroys parts of the stack. We loose the direct access property of an array, but we only need to show that we can simulate the functionality in principle, not efficiently.

We do not check whether i is in the range $0, \ldots, m-1$ to keep the program simpler. In fact, we will even simulate dynamic arrays, that is, we can change the size of the array during runtime. It is easily checked that

$\langle 0, \langle 0, \ldots \langle 0, 0\rangle \ldots\rangle\rangle = 0$. Therefore, we can interpret the initial value 0 as an infinite sequence of 0. So the dynamic part comes essentially for free.

Program 7 Extracting the ith element from A

Output: $A[i]$ is returned in x

```
1: B := A;
2: for i do
3:    pop(B)
4: od;
5: x := top(B)
```

Next, we implement the operation $A[i] := b$. The implementation is very similar to the previous program, instead of returning the top element in line 5, we replaced it by b. Furthermore, we have to restore the stack in the end, therefore, we use a second stack to store the elements that were popped from A before.

Program 8 Simulating $A[i] := b$

```
 1: B = 0;
 2: for i do
 3:    push(B, top(A));
 4:    pop(A)
 5: od;
 6: pop(A);
 7: push(A, b);
 8: for i do
 9:    push(A, top(B));
10:    pop(B)
11: od;
```

8.7 Exercises

Basic exercises

Exercise 8.5 *Show how to simulate the if-then-else statement in FOR.*

Exercise 8.6 *The following construct is also useful. Explain how to simulate them in simple WHILE.*

- ***Input:*** $v_1, \ldots, v_s$ *declares* $v_1, \ldots, v_s$ *as the input variables.*

Intermediate exercises

Exercise 8.7 *Explain how to simulate the following statement in simple WHILE:*

- ***return*** *x leaves the current program immediately and the value of x is the output of the program.*

Here is another function that we could use as a pairing function. It is injective but not surjective.

Exercise 8.8 *Let k be some constant. Let $p_1, \ldots, p_k$ be different prime numbers.*

1. *Show that the mapping g given by*

$$\begin{aligned} \mathbb{N}^k &\to \mathbb{N} \\ (x_1, \ldots, x_k) &\mapsto p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k} \end{aligned}$$

   is an injective mapping.

2. *Show that g is FOR computable.*

3. *Show that there is a FOR program that decides whether a given $y \in \mathbb{N}$ is in $\operatorname{im} g$.*

4. *Show that there is a FOR program that given $y \in \operatorname{im} g$ computes the unique $x_1, \ldots, x_k$ such that $g(x_1, \ldots, x_k) = y$.*

II Ackermann function

Chapters that are numbered with Roman numbers instead of Arabic ones are for your personal entertainment only. They are not an official part of the lecture, in particular, not relevant for any exams. But reading them does not hurt either ...

It is clear that there are functions that are WHILE computable but not FOR computable, since FOR programs can only compute total functions but WHILE programs can compute partial ones. Are there *total* functions that are WHILE computable but not FOR computable? I.e. are WHILE loops more powerful than FOR loops? The answer is affirmative.

II.1 Definition

The *Ackermann function* is a WHILE computable but not FOR computable total function, which was first published in 1928 by Wilhelm Ackermann, a student of David Hilbert. The so called Ackermann-Péter-Function, which was defined later (1955) by Rózsa Péter and Raphael Robinson has only two variables (instead of three).

The Ackermann function is maybe the simplest example of a total function that is WHILE computable but not FOR computable, providing a counterexample to the belief in the early 1900s that every WHILE computable function was also FOR computable. (At that time, the two concepts were called recursive and primitive recursive.) It grows faster than an exponential function, or even a multiple exponential function. In fact, it grows faster than most people (including me) can even imagine.

The Ackermann function is defined recursively for non-negative integers x, y by

$$\begin{array}{lcll} a(0,y) & = & y+1 & \\ a(x,0) & = & a(x-1,1) & \text{for } x>0 \\ a(x,y) & = & a(x-1,a(x,y-1)) & \text{for } x,y>0 \end{array}$$

Lemma II.1 *a is a total function, i.e, $a(x,y)$ is defined for all $x,y \in \mathbb{N}$.*

Proof. We prove by induction on x that $a(x,y)$ is defined for all $x,y \in \mathbb{N}$.
Induction base: Starting with the induction base $x=0$ gives us $a(0,y)=y+1$ for all y.
Induction step: The induction step $x-1 \to x$ again is a proof by induction, now on y. We start with $y=0$. By definition, $a(x,0) = a(x-1,1)$.

The right-hand side is defined by the induction hypothesis for $x-1$. For the induction step $y-1 \to y$ note that $a(x,y) = a(x-1, a(x,y-1))$ by definition. The right-hand side is defined by the induction hypotheses for $x-1$ and also $y-1$. ■

Lemma II.2 *a is WHILE computable.*

Proof. We prove this by constructing a WHILE program that given x and y, computes $a(x,y)$. We use a stack S for the computation. Program 9 will compute the value of a with the top two elements of the stack as the arguments. It first pushes the two inputs x and y on the stack. Then it uses the recursive rules of a to compute the value of a. This might push new elements on the stack, but since a is total, the stack will eventually be consisting of a sole element, the requested value. The function size(S) returns the number of elements in the stack, it is implemented by $\pi_1(S)$. ■

II.2 Some closed formulas

In this section, we keep x fixed and consider a as a function in y. For small x, we can express $a(x,.)$ in closed form. For $x=1$, we get by applying the appropriate rules for a:

$$\begin{aligned} a(1,y) &= a(0, a(1,y-1)) \\ &= a(1,y-1)+1 \\ &= a(0,a(1,y-2))+1 \\ &= a(1,y-2)+2 \\ &\vdots \\ &= a(1,0)+y \\ &= a(0,1)+y \\ &= y+2. \end{aligned}$$

For $x=2$, we have

$$\begin{aligned} a(2,y) &= a(1, a(2,y-1)) \\ &= a(2,y-1)+2 \\ &\vdots \\ &= a(2,0)+2y \\ &= a(1,1)+2y \\ &= 2y+3. \end{aligned}$$

Here we used the closed formula $a(1, y) = y + 2$.

Finally, for $x = 3$, we obtain

$$\begin{aligned}
a(3, y) &= a(2, a(3, y-1)) \\
&= 2 \cdot a(3, y-1) + 3 \\
&= 2 \cdot a(2, a(3, y-2)) + 3 \\
&= 2 \cdot (2 \cdot a(3, y-2) + 3) + 3 \\
&= 2^2 \cdot a(3, y-2) + 3 \cdot (1 + 2) \\
&= 2^2 \cdot (2 \cdot a(3, y-3) + 3) + 3 \cdot (1 + 2) \\
&= 2^3 \cdot a(3, y-3) + 3 \cdot (1 + 2 + 2^2) \\
&\vdots \\
&= 2^y \cdot a(3, 0) + 3 \cdot \sum_{k=0}^{y-1} 2^i \\
&= 2^y \cdot a(3, 0) + 3 \cdot (2^y - 1) \\
&= 2^y \cdot a(2, 1) + 3 \cdot 2^y - 3 \\
&= 2^y \cdot 5 + 3 \cdot 2^y - 3 \\
&= 2^{y+3} - 3.
\end{aligned}$$

So $a(1, y)$ is addition of 2, $a(2, y)$ is (essentially) multiplication with 2, and $a(3, y)$ is (essentially) exponentiation with base 2.

Exercise II.1 *Show that* $a(4, y) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3} - 3$.

II.3 Some useful facts

In this section, we show some monotonicity properties of a.

Lemma II.3 *The function value is strictly greater then its second argument, i.e.,* $\forall x, y \in \mathbb{N}$

$$y < a(x, y).$$

Proof. Again, we show this by induction on x.
Induction base: For $x = 0$, we just use the definition of a which yields

$$a(0, y) = y + 1 > y.$$

Induction step: The induction step $x - 1 \to x$ is again shown by an inner induction on y. So we start with $y = 0$:

$$a(x, 0) = a(x - 1, 1) > 1 > 0,$$

where the first inequality follows from the induction hypothesis for x. The induction step for the inner induction is shown as follows:

$$a(x,y) = a(x-1, a(x,y-1)) \;>\; a(x,y-1) \;>\; y-1. \tag{II.1}$$

$a(x-1, a(x,y-1))$ is strictly greater than $a(x,y-1)$ because of the induction hypothesis for x. But this is even strictly greater than $y-1$ because of the induction hypothesis for y.

We need to prove that $a(x,y) > y$. But in (II.1), there stands a "$>$" twice. Thus (II.1) even implies $a(x,y) > y$. ∎

Lemma II.4 *The Ackermann function is strictly monotonically increasing in the second argument, i.e.,* $\forall x, y \in \mathbb{N}$

$$a(x,y) < a(x,y+1).$$

Proof. We consider two cases. For $x = 0$ we have $a(0,y) = y+1$ which is less than $y+2 = a(0,y+1)$. So we get $a(0,y) < a(0,y+1)$.

For $x > 0$ we see from Lemma II.3 that $a(x,y) < a(x-1, a(x,y))$ which equals $a(x,y+1)$ by the definition of the Ackermann function. ∎

Lemma II.5 *For all* $x, y \in \mathbb{N}$, $a(x,y+1) \leq a(x+1,y)$.

Proof. By induction on y.
Induction base: For $y = 0$ the equation $a(x,1) = a(x+1,0)$ follows from the definition of a.
Induction step: Consider the induction step $y-1 \to y$. By Lemma II.3, we know that $y < a(x,y)$. Thus $y+1 \leq a(x,y) \leq a(x+1,y-1)$ because of the induction hypothesis. Lemma II.4 allows us to plug the inequality $y+1 \leq a(x+1,y-1)$ into the second argument yielding

$$a(x,y+1) \leq a(x, a(x+1,y-1)) = a(x+1,y). \qquad ∎$$

Lemma II.6 *The Ackermann function is strictly monotonically increasing in the first argument as well, i.e.,* $\forall x, y \in \mathbb{N}$,

$$a(x,y) < a(x+1,y).$$

Proof. Using Lemma II.4 first and then Lemma II.5, we obtain

$$a(x,y) < a(x,y+1) \leq a(x+1,y). \qquad ∎$$

II.4 The Ackermann function is not FOR computable

Let P be a FOR program that uses the variables $x_0, \ldots, x_\ell$. Assume that these variables are initialized with the values $v_0 \ldots v_\ell$ and that after the execution of P, the variables contain the values $v'_0 \ldots v'_\ell$. We now define a function $f_P(n)$ that essentially measures the size of the output that a FOR program can produce (in terms of the size of the input). f_P is defined via

$$f_P(n) = \max \left\{ \sum_{i=0}^{\ell} v'_i \;\middle|\; \sum_{i=0}^{\ell} v_i \leq n \right\}.$$

In other words, $f_P(n)$ bounds the sum of the values of $x_0, \ldots, x_\ell$ after the execution of P in terms of the sum of the values of $x_0, \ldots, x_\ell$ before the execution of P.

Lemma II.7 *For every FOR program P there is a $k \in \mathbb{N}$ such that*

$$f_P(n) < a(k, n).$$

Proof. By structural induction. We can assume w.l.o.g. that the simple instructions of the FOR program are either $x_i{+}{+}$ or $x_i{-}{-}$ or $x_i := 0$, cf. Exercise 7.7

Furthermore, we assume that for every FOR loop **for** x_i **do** Q **od** in P, x_i does not appear in Q. If it does, we can replace the loop first by $x_k := x_i$; **for** x_k **do** P_1 **od** where x_k is an unused variable. And of course, since we have only a restricted set of simple statements, we have to replace the assignment $x_k := x_i$ by something else, too. For the moment, let $\hat{P}$ be the resulting program. Let x_i be an original variable of P. After the execution of $\hat{P}$, the value of x_i is the same as the value of x_i after the execution of P. Therefore, $f_P(n) \leq f_{\hat{P}}(n)$ and it suffices to bound $f_{\hat{P}}(n)$.
Induction base: Let $P = x_i{+}{+}$ be a FOR program. If the sum of the values is bounded by n before the execution of P, then it is bounded by $n+1$ after the execution. Thus

$$f_P(n) \leq n + 1 < a(1, n).$$

If $P = x_i := 0$ or $P = x_i{-}{-}$, then the size of the ouput cannot be larger than the size of the output. Hence

$$f_P(n) \leq n < a(0, n),$$

too. This finishs the induction base.
Induction step: For the induction step, let $P = P_1; P_2$ be a FOR program. From the induction hypothesis we know that $f_{P_i}(n) < a(k_i, n)$ for constants k_i and $i \in \{1, 2\}$. After the execution of P_1, the sum of the values of the

variables is bounded by $a(k_1, n)$ which is also a bound on the sum of the values of the variables before the execution of P_2. Altogether,

$$\begin{array}{rcll} f_P(n) & = & f_{P_2}(f_{P_1}(n)) & \\ & < & a(k_2, a(k_1, n)) & \\ & < & a(k_3, a(k_3, n)) & \text{by monotonicity with } k_3 = \max\{k_1, k_2\} \\ & < & a(k_3, a(k_3 + 1, n)) & \\ & = & a(k_3 + 1, n + 1) & \text{per definition} \\ & < & a(k_3 + 2, n). & \end{array}$$

Now, let $P = \textbf{for } x_i \textbf{ do } P_1 \textbf{ od}$ be a FOR program. Recall that x_i does not occur in P_1. From the induction hypothesis, we get $f_{P_1}(n) < a(k_1, n)$. Fix a tuple $\nu_0, \ldots, \nu_\ell$ for which the maximum is attained in the definition of f_P. Let $m := \nu_i$ be the value of x_i in this tuple. We distinguish three cases:

$m = 0$: In this case, $f_P(n) = n < a(0, n)$, since the loop is not executed at all.

$m = 1$: Here, $f_P(n) \leq f_{P_1}(n) < a(k_1, n)$, since the loop is executed only once.

$m > 1$: Here we have,

$$\begin{aligned} f_P(n) &= \underbrace{f_{P_1} \circ \ldots \circ f_{P_1}}_{m \text{ times}}(n - m) + m \\ &< a(k_1, \underbrace{f_{P_1} \circ \ldots \circ f_{P_1}}_{m-1 \text{ times}}(n - m)) + m \\ &\leq a(k_1, \underbrace{f_{P_1} \circ \ldots \circ f_{P_1}}_{m-1 \text{ times}}(n - m)) + m - 1 \\ &\vdots \\ &\leq a(k_1, a(k_1, \ldots a(k_1, n - m))) \\ &< a(\underbrace{k_1, a(k_1, \ldots,}_{m \text{ times}} a(k_1 + 1, n - m))) \\ &< a(k_1 + 1, n). \qquad \blacksquare \end{aligned}$$

Theorem II.8 *The Ackerman function a is not FOR computable.*

Proof. Assume that a was FOR computable, then $\hat{a}(k) = a(k, k)$ is FOR computable as well. Let P be a FOR program for $\hat{a}$. Lemma II.7 tells us that there is a k such that

$$\hat{a}(k) \leq f_P(k) < a(k, k) = \hat{a}(k),$$

a contradiction. This proves that a is not FOR computable. $\blacksquare$

Program 9 Ackermann function

Input: x, y

```
 1: push(S, x);
 2: push(S, y);
 3: while size(S) > 1 do
 4:    y := pop(S);
 5:    x := pop(S);
 6:    if x = 0 then
 7:       push(S, y + 1);
 8:    else
 9:       if y = 0 then
10:          push(S, x − 1);
11:          push(S, 1);
12:       else
13:          push(S, x − 1);
14:          push(S, x);
15:          push(S, y − 1);
16:       fi
17:    fi
18: od
19: x_0 := pop(S);
```

9 Gödel numberings

In this chapter, we address two fundamental questions. The first one is: How many WHILE programs are there? And the second one is: How can we feed a WHILE program into another WHILE program as an input?

There are certainly infinitely many WHILE programs: $x_0 := 0$ is one, $x_0 := 0; x_0 := 0$ is another, $x_0 := 0; x_0 := 0; x_0 := 0$ is a third one; I guess you are getting the idea. But there are different "sorts of infinite".

Definition 9.1 *A set S is* countable *if there exists an injective*[1] *function $S \to \mathbb{N}$. If there is a bijection $S \to \mathbb{N}$, then S is called* countably infinite.

Exercise 9.1 *Prove that if there is an injective function $f : S \to \mathbb{N}$ such that* im f *is infinite, then there is a bijective function $S \to \mathbb{N}$.*

Exercise 9.2 *Prove the following: If S is countable, then S is finite or countably infinite.*

Recall that the pairing function $\langle ., . \rangle$ is a bijection from $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Thus $\mathbb{N} \times \mathbb{N}$ is countable, too.

Exercise 9.3 *1. Prove that $\mathbb{Q}$ is countable.*

2. Prove that if A and B are countable, so is $A \times B$.

We will show that the set of all WHILE programs is countably infinite. By assigning each WHILE program a natural number in a unique way, we can feed them into other WHILE programs, too. For just proving that the set of all WHILE programs is countable, we can use any injective function. But for the purpose of feeding WHILE programs into WHILE programs, this function should also have some nice properties.

We will construct an bijective function göd : $\mathcal{W} \to \mathbb{N}$, that is, different WHILE programs get different numbers. But this is not enough, we also need the following property:

- There is a WHILE program U that given $i, x \in \mathbb{N}$ computes $\varphi_P(x)$ where $P = \text{göd}^{-1}(i)$.

Remark 9.2 *In some books, the mapping* göd *is only required to be injective. In this case, we also need a second WHILE program C, that given $i \in \mathbb{N}$, outputs* 1 *if $i \in$* im göd *and* 0 *otherwise, that is, C decides* im göd. *Since our mapping will be bijective, we can set $C = x_0 := 1$.*

[1] Recall that injective includes that the function is total

Furthermore, given $i \in \mathbb{N}$ it is "easy" to find the WHILE program P with $\text{göd}(P) = i$. "Easy" is not formally specified here. We could extend the WHILE language and add a print command that can print fixed strings and the content of variables. Then it is quite easy to print P if i is given.

Such a mapping göd is usually called *Gödel numbering*, named after the mathematician Kurt Gödel. The term "Gödel numbering" is used for many different things, so be careful. (The existence of U might not be included in the definition.) Gödel numberings allow us to treat programs as data. WHILE programs can now analyze and manipulate other WHILE programs.

Excursus: Kurt Gödel

Kurt Gödel (born 1906 in Brno, Austria–Hungary (now Czech Republic) and died 1978 in Princeton, USA) was a mathematician and logician.

He is best known for his incompleteness theorem which roughly says that in any self-consistent recursive axiomatic system powerful enough to describe the arithmetics of $\mathbb{N}$, there are theorems that are true but cannot be deduced from the axioms. (More one this later.) This destroyed David Hilbert's dream that everything in mathematics can be proven from a correctly-chosen finite system of axioms

In Princeton, Kurt Gödel became a close friend of Albert Einstein. Gödel, to put it nicely, was a rather complicated person.

We define a concrete Gödel numbering göd for WHILE programs that we will use throughout this lecture. göd will be defined inductively. We will use our pairing function $\langle ., . \rangle$. But we need a second pairing function, too.

Fact 9.3 *The map given by* $(r, m) \mapsto \langle r, m \rangle_5 := 5m + r$ *is a bijection* $\{0, 1, 2, 3, 4\} \times \mathbb{N} \to \mathbb{N}$.

We denote the projections that invert $\langle ., . \rangle_5$ by π_1 and π_2, since there will usually no confusion with the projections that invert $\langle ., . \rangle$. Note that $\pi_1(n) = n \bmod 5$ and $\pi_2(n) = \lfloor n/5 \rfloor$.

We start with the simple statements:

1. The statement $x_i := x_j + x_k$ is encoded by $\langle 0, \langle i, \langle j, k \rangle \rangle \rangle_5$.
2. The statement $x_i := x_j - x_k$ is encoded by $\langle 1, \langle i, \langle j, k \rangle \rangle \rangle_5$.
3. The statement $x_i := c$ is encoded by $\langle 2, \langle i, c \rangle \rangle_5$.

The while loop and the concatenation of programs is encoded as follows:

1. If $P =$ **while** $x_i \neq 0$ **do** P_1 **od**, then $\text{göd}(P) = \langle 3, \langle i, \text{göd}(P_1) \rangle \rangle_5$.
2. If $P = [P_1; P_2]$, then $\text{göd}(P) = \langle 4, \langle \text{göd}(P_1), \text{göd}(P_2) \rangle \rangle_5$.

Remark 9.4 *Instead of* $\langle ., . \rangle_5$*, we could also use the function* $\langle ., . \rangle$*. Then* göd *would only be injective, which would be fine for our purposes. Being bijective spares us some case distinctions.*

Remark 9.5 *The Gödelnumbers of* $[[P_1; P_2]; P_3]$ *and* $[P_1; [P_2; P_3]]$ *are different, but this is o.k., since syntactically, they are different programs.*

Lemma 9.6 göd *is well-defined.*

Proof. The proof is again by structural induction.
Induction base: Obviously, göd(P) is well-defined for all $P \in \mathcal{W}_0$.
Induction step: Now assume that $P \in \mathcal{W}_n$ for some n. Then either $P =$ **while** $x_i \neq 0$ **do** P_1 **od** or $P = [P_1; P_2]$ for some $P_1, P_2 \in \mathcal{W}_{n-1}$. By the induction hypothesis, göd(P_1) and göd(P_2) are both well-defined. Thus göd$(P) = \langle 3, \langle i, \text{göd}(P_1)\rangle\rangle_5$ or göd$(P) = \langle 4, \langle \text{göd}(P_1), \text{göd}(P_2)\rangle\rangle_5$, respectively, are both well-defined, too. ∎

Lemma 9.7 göd *is injective.*

Proof. We show the statement that for all n, göd$(P) =$ göd(Q) implies $P = Q$ for all $P, Q \in \mathcal{W}_n$. From this, the assertion of the lemma follows. (So we show that for all n, göd restricted to $\mathcal{W}_n$ is injective. Since $\bigcup_{n\in\mathbb{N}} \mathcal{W}_n = \mathcal{W}$, this implies the injectivity of göd : $\mathcal{W} \to \mathbb{N}$.)
Induction base: We prove the statement for all programs in $\mathcal{W}_0$. Assume that göd$(P) =$ göd(Q) for some $P, Q \in \mathcal{W}_0$. In particular, $t := \pi_1(\text{göd}(P)) = \pi_1(\text{göd}(Q))$. Since $P \in \mathcal{W}_0$, $t \in \{0, 1, 2\}$. Assume $t = 0$. Then $\pi_2(\text{göd}(P)) = \langle i, \langle j, k\rangle\rangle$ and i, j, k are unique, since $\langle ., \langle ., .\rangle\rangle$ is a bijection $\mathbb{N}^3 \to \mathbb{N}$. Since göd$(P) =$ göd(Q) also implies $\pi_2(\text{göd}(P)) = \pi_2(\text{göd}(Q))$, we get $P = Q$. The cases $t = 1, 2$ are treated in the same way.
Induction step: Now assume that göd$(P) =$ göd(Q) and assume that $P \in \mathcal{W}_n \setminus \mathcal{W}_{n-1}$ for some $n > 0$ and $Q \in \mathcal{W}_n$.[2] Since $n > 0$, göd(P) is either $\langle 3, \langle i, \text{göd}(P_1)\rangle\rangle_5$ or $\langle 4, \langle \text{göd}(P_1), \text{göd}(P_2)\rangle\rangle_5$ for some programs $P_1, P_2 \in \mathcal{W}_{n-1}$. We only treat the case göd$(P) = \langle 3, \langle i, \text{göd}(P_1)\rangle\rangle_5$, the other case is an exercise. göd$(P) =$ göd(Q) in particular implies that $\pi_1(\text{göd}(P)) = \pi_1(\text{göd}(Q))$. This shows that göd$(Q) = \langle 3, \langle j, \text{göd}(Q_1)\rangle\rangle_5$. But also $\pi_2(\text{göd}(P)) = \pi_2(\text{göd}(Q))$, i.e, $\langle i, \text{göd}(P_1)\rangle = \langle j, \text{göd}(Q_1)\rangle$. But since $\langle ., .\rangle$ is a bijection, this means that $i = j$ and göd$(P_1) =$ göd(Q_1). By the induction hypothesis, $P_1 = Q_1$. Thus $P = Q$. ∎

Corollary 9.8 *The set of all WHILE programs* $\mathcal{W}$ *is countable.*

Exercise 9.4 *Fill in the missing part in the proof of Lemma 9.7.*

Lemma 9.9 göd *is surjective.*

[2] We know that $P, Q \in \mathcal{W}_n$. We can assume that one of them, say P, is not in $\mathcal{W}_{n-1}$, otherwise we would be done by the induction hypothesis. The other one could still be contained in $\mathcal{W}_{n-1}$, though.

Proof. We prove by induction on n that for all n, there is a WHILE program P with $\text{göd}(P) = n$.
Induction base: We have $0 = \langle 0, \langle 0, \langle 0, 0\rangle\rangle\rangle = \text{göd}(x_0 := x_0 + x_0)$.
Induction step: Let $n = \langle r, m\rangle_5$ with $0 \leq r \leq 4$. We distinguish five cases, depending on r.

We start with $r = 0$. There are numbers i, j, k such that $\langle i, \langle j, k\rangle\rangle = m$, since our pairing function $\langle ., .\rangle$ is surjective. Therefore

$$n = \langle 0, m\rangle_5 = \langle 0, \langle i, \langle j, k\rangle\rangle\rangle_5 = \text{göd}(x_i := x_j + x_k).$$

The cases $r = 1, 2$ are treated similarly, we can write $n = \text{göd}(x_i := x_j - x_k)$ for appropriate numbers i, j, k or $n = \text{göd}(x_i := c)$ for appropriate numbers i, c, respectively.

If $r = 3$, then we choose numbers i, s such that $m = \langle i, s\rangle$. Our pairing function fulfills $\langle x, y\rangle \geq x, y$ for all x, y. Since $r = 3$, $n > m \geq s$. Therefore, we can apply the induction hypothesis and write $s = \text{göd}(Q)$ for some WHILE program Q. Thus

$$n = \langle 3, \langle i, \text{göd}(Q)\rangle\rangle_5 = \text{göd}(\text{while } x_i \neq 0 \text{ do } Q \text{ od}).$$

The case $r = 4$ can be treated in the same way, we write $m = \langle x_1, x_2\rangle$ and obtain $n = \text{göd}([Q_1; Q_2])$, where $x_i = \text{göd}(Q_i)$, $i = 1, 2$. ∎

Excursus: Programming systems

The mapping göd associates with every $i \in \mathbb{N}$ a function $\varphi_{\text{göd}^{-1}(i)}$. Instead of $\varphi_{\text{göd}^{-1}(i)}$, we will often write φ_i.

Definition 9.10 *1. A sequence $(\psi_i)_{i\in\mathbb{N}}$ is called a* programming system *if the set of all ψ_i is precisely* R*, the set of all WHILE computable functions.*

2. It is universal*, if the programming system has a universal program, i.e, there is an index u such that $\psi_u(\langle j, x\rangle) = \psi_j(x)$ for all $j, x \in \mathbb{N}$.*

3. A universal programming system is called acceptable *if there is an index c such that $\psi_{\psi_c(\langle j,k\rangle)} = \psi_j \circ \psi_k$.*

The sequence $(\varphi_i)_{i\in\mathbb{N}}$ of WHILE computable functions is an acceptable programming system. It is certainly a programming system and we will see soon that it is universal by constructing a universal program U. To see that it is acceptable, note that $\langle 4, \langle k, j\rangle\rangle$ is the Gödel number of $\varphi_j \circ \varphi_k$, if $j, k \in \mathbb{N}$. c will be the Gödel number of the program computing $\langle 4, .\rangle$

There are other programming systems and we will see some of them. Everything that we prove in the remainder of this part is valid for every acceptable programming system. For instance, the set of all functions computed by JAVA programs is an acceptable programming systems. (We have to identify ASCII texts with natural numbers somehow. We will formally do this in later chapters.)

---

10 Diagonalization

In this chapter we will answer the question whether there is a function $\mathbb{N} \to \mathbb{N}$ that is *not* computable by a WHILE program. (This will be a mathematician's answer. It will be true but absolutely useless. ;-) "Useless" here means that the function we construct is not natural in some sense. We will prove in later chapters that there are indeed many important and natural functions that are not WHILE computable.)

10.1 Proof by "counting"

Basically, we will show that the set of all total functions $\mathbb{N} \to \mathbb{N}$ is not countable. Even the set of all functions $\mathbb{N} \to \{0,1\}$ is not countable.

Theorem 10.1 *The set of all total functions $\mathbb{N} \to \{0,1\}$ is not countable.*

Proof overview: The proof will use a technique that is called *Cantor's diagonal argument*. We assume that the set of all total functions $\mathbb{N} \to \{0,1\}$, call it F, is countable. Then there is a bijection n between F and $\mathbb{N}$, i.e., each function in $f \in F$ gets a "number" $n(f)$. We construct a total function $c : \mathbb{N} \to \{0,1\}$ that differs from every $f \in F$ on the *input* $n(f)$. $c \in F$ by construction, on the other hand it differs from every $f \in F$ on some input. This is a contradiction.

Proof. Assume that F is countable and let $n : F \to \mathbb{N}$ be a bijection. (F is certainly infinite and by Exercise 9.1, we can assume that n is bijective.) Let f_i be the function in F that is mapped onto i by n, i.e., $n(f_i) = i$ for all $i \in \mathbb{N}$.

We arrange the values of the functions f_i in a 2-dimensional tabular. The ith row contains the values of f_i and the jth column contains the values of all functions on input j. This means that the entry in position (i,j) contains the value $f_i(j)$ (see Figure 10.1).

Now we define the function c as follows: $c(i) = 1 - f_i(i)$ for all $i \in \mathbb{N}$. In other words,

$$c(i) = \begin{cases} 0 & \text{if } f_i(i) = 1, \\ 1 & \text{if } f_i(i) = 0. \end{cases}$$

c differs from the f_i's in the entries that are underlined in the table in Figure 10.1. Clearly $c \in F$. But this means that there is an index i_0 such

	0	1	2	3	...
0	$\underline{f_0(0)}$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
1	$f_1(0)$	$\underline{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
2	$f_2(0)$	$f_2(1)$	$\underline{f_2(2)}$	$f_2(3)$	...
3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$\underline{f_3(3)}$	...
⋮	⋮	⋮	⋮	⋮	⋱

Figure 10.1: The diagonalization scheme

that $f_{i_0} = c$, since $\{f_i \mid i \in \mathbb{N}\} = F$. In particular,

$$f_{i_0}(i_0) = c(i_0) = 1 - f_{i_0}(i_0).$$

But this is a contradiction since $f_{i_0}(i_0)$ is a natural number and the equation $x = 1 - x$ has no integral solution. ■

Corollary 10.2 *There is a total function $\mathbb{N} \to \{0,1\}$ that is not WHILE computable.*

Proof. The proof is by contradiction: If every function from F, the set of all total functions $\mathbb{N} \to \{0,1\}$ was WHILE computable, then the image of the mapping given by $P \mapsto \varphi_P$ would contain F. But this means that there is an injective mapping i_1 from F to some subset of $\mathcal{W}$. Since the set $\mathcal{W}$ of all WHILE programs is countable, there is an injective mapping i_2 from $\mathcal{W} \to \mathbb{N}$. The composition $i_2 \circ i_1$ is an injective mapping from F to $\mathbb{N}$. This means that F is countable, a contradiction. ■

Since the characteristic function of a subset of $\mathbb{N}$ is a function $\mathbb{N} \to \{0,1\}$, we get the following corollary.

Corollary 10.3 *There is a subset of $\mathbb{N}$ that is not recursive.*

Excursus: Cantor's diagonal argument

Georg F. L. P. Cantor (born 1845 in St. Petersburg, died 1918 in Halle), was a German mathematician. He is known as one of the creators of set theory.

We saw two accomplishments of Cantor in this lecture: The proof that $\mathbb{N}^2$ is again countable and the diagonal argument that show that the set of all total functions $\mathbb{N} \to \{0,1\}$ is not countable.

The following conversation between Dedekind and Cantor is reported:
Dedekind: "Eine Menge stelle ich mir vor wie einen Sack mit Dingen darin."
Cantor (in theatric pose): "Eine Menge stelle ich mir vor wie einen Abgrund."
Cantor should be right.

10.2 Explicit construction

The proof of the existence of a characteristic function that is not WHILE computable was indirect and used the fact that there are more characteristic functions than WHILE programs. Basically the same proof also yields a direct construction of a characteristic function that is not WHILE computable. This construction will be explicit, i.e., we can precisely say how the function looks like. The function c that we will construct has the property that for all $i \in \mathbb{N}$, $c(i)$ is defined iff $\varphi_P(i)$ is not defined, where $P = \text{göd}(i)$. In other words, $c(i)$ is defined whenever the WHILE program with Gödel number i does not halt on i. This c is at least "semi-natural". We will see soon how to prove that natural tasks, like verification, are not computable.

Overview over alternative proof of Corollary 10.2: We will use the same diagonalization scheme as in Figure 10.1. The construction becomes explicit, since we do not use a hypothetical enumeration of all characteristic functions but an enumeration of all WHILE programs that we already constructed.

Alternative proof of Corollary 10.2. We constructed an bijective mapping göd from the set of all WHILE programs to $\mathbb{N}$ in Chapter 9. We now define a sequence of functions $f_0, f_1, f_2, \ldots$ by

$$f_i(j) = \varphi_{\text{göd}^{-1}(i)}(j).$$

for all i and j. That means, when P is the WHILE program such that $\text{göd}(P) = i$, then f_i will be the function φ_P computed by P.

Now define the function c by

$$c(n) = \begin{cases} 1 & \text{if } f_n(n) = 0 \text{ or is undefined} \\ 0 & \text{otherwise} \end{cases}$$

for all $n \in \mathbb{N}$. There is no WHILE program P that can compute c, because if $\text{göd}(P) = i$, then c differs from φ_P at the input i. ■

Remark 10.4 *c essentially is the characteristic function of the set of all Gödel numbers i such that* $\text{göd}^{-1}(i)$ *either does not halt on i or returns* 0.

Excursus: Programming systems II

Corollary 10.2 holds for all programming systems. All that we used is that there is a mapping $i \mapsto \psi_i$ such that the image of this mapping is R.

10.3 Exercises

Intermediate exercise

Exercise 10.1 *Show that for any nonempty set S, there is no bijection between S and its power set $\mathcal{P}(S)$.*

11 A universal WHILE program

In this chapter, we will construct the universtal WHILE program U for our function göd. Assume that we are given an index $g \in \mathbb{N}$, i.e., an encoding g of a WHILE program P and an $m \in \mathbb{N}$. U now has to simulate P on input m with only a fixed number of variables. The program P has a fixed number of variables, too, but since U has to be capable of simulating every WHILE program, there is no a priori bound on the number of variables in P. Thus U will use an array X to store the values of the variables of P. Luckily, we do already know how to simulate arrays in WHILE (and even FOR). Let ℓ be the largest index of a variable that occurs in P. Then an array of length $\ell + 1$ is sufficient to store all the values. It is not too hard to extract this number ℓ given g. But since any upper bound on ℓ is fine too, we just use an array of length g in U. g is an upper bound on ℓ because of Exercise 11.1 (and the way we constructed göd).

Exercise 11.1 *Show that for all $j, k \in \mathbb{N}$, $\langle j, k\rangle \geq \max\{j, k\}$.*

A simple statement is encoded as $\langle 0, \langle i, \langle j, k\rangle\rangle\rangle_5$ (addition), $\langle 1, \langle i, \langle j, k\rangle\rangle\rangle_5$ (subtraction), or $\langle 2, \langle i, c\rangle\rangle_5$ (initialization with constant). Using π_1, we can project onto the first component of these nested pairs and find out whether the statement is an addition, subtraction, or initialization with a constant. The result that we get by application of π_2 then gives us the information about the variables and/or constants involved. Program 10 shows how to perform the addition. X stores the array that we need to simulate. When we plug this routine into U, we might have to rename variables.

Program 10 Subroutine for addition

Input: $\langle i, \langle j, k\rangle\rangle$ stored in variable x_2

1: $x_3 := \pi_1(x_2)$;
2: $x_4 := \pi_1(\pi_2(x_2))$;
3: $x_5 := \pi_2(\pi_2(x_2))$;
4: $X[x_3] := X[x_4] + X[x_5]$

Exercise 11.2 *Write the corresponding programs for subtraction and initialization with a constant.*

More problematic are while loops and concatenated programs. We will use a *stack* S to keep track of the program flow. Luckily, we know how to implement a stack in WHILE (and even FOR).

Program 11 is our universal WHILE program. (Look at the code for a *while*. Then try to imagine you had to write a universal C++ program. Next, thank your professor.) There are five important variables in it whose meaning we explain below:

- X**:** the array that stores the values of the variables in the program $P := \text{göd}^{-1}(g)$.
- S**:** the stack that stores (encodings of) pieces of P to be executed later
- *cur*: a variable that stores the piece of P that we are right now simulating
- *term*: is 0 if our simulation terminated and 1 otherwise.
- *type*: stores the type (0 to 4) of the current statement that we are simulating

In lines 1 to 5, we initalize X and store the input m in $X[0]$. Our current program that we work on is g. We will exit the while loop if we finish the simulation of P. *cur* will always be of the form

- $\langle 0, \langle i, \langle j, k\rangle\rangle\rangle_5$ (addition)
- $\langle 1, \langle i, \langle j, k\rangle\rangle\rangle_5$ (subtraction)
- $\langle 2, \langle i, c\rangle\rangle_5$ (initialization)
- $\langle 3, \langle i, \text{göd}(P_1)\rangle\rangle_5$ (while loop)
- $\langle 4, \langle \text{göd}(P_1), \text{göd}(P_2)\rangle\rangle_5$ (concatenation)

In line 6, we set $type := \pi_1(cur)$. This value is between 0 to 4. If $type \in \{0, 1, 2\}$[1], then we just have to simulate an addition, subtraction, or initialization. This is easy. The next two cases are far more interesting.

If $type = 3$, then $cur = \langle 3, \langle i, \text{göd}(P_1)\rangle\rangle$. So we have to simulate a while loop **while** $x_i \neq 0$ **do** P_1 **od**. In line 18, we check whether the condition $x_i \neq 0$ is fulfilled. If not, then we do not enter the while loop. If yes, then we do the following: First we simulate P_1 once and then we simulate the while loop again. Therefore, we push *cur* (which equals $\langle 3, \langle i, \text{göd}(P_1)\rangle\rangle$) and $\pi_2(\pi_2(cur))$ (which equals $\text{göd}(P_1)$) on the stack.

If $type = 4$, then $cur = \langle 4, \langle \text{göd}(P_1), \text{göd}(P_2)\rangle\rangle$. We push $\pi_2(\pi_2(cur))$ (which is $\text{göd}(P_2)$) on the stack and then $\pi_1(\pi_2(cur))$ (which is $\text{göd}(P_1)$). In this way, we will first execute P_1 on top of the stack and then P_2.

At the end of the while loop, we check whether the stack is empty. If yes, then our simulation has finished and we just have to copy $X[0]$, the output of the simulation, into x_0. If the stack is not empty, we pop the next piece of program from the stack, store it into *cur*, and go on with the simulation.

[1] That is not 100% correct; what we mean is that the *value* of *type* is in $\{0, 1, 2\}$. We will use this sloppy notation often in the remainder of this chapter.

Program 11 Universal WHILE program U

Input: Gödel number g of P, input m of P
Output: $\varphi_P(m)$ if P terminates on m. Otherwise U does not terminate.

1: $X := 0$; {Sets all entries of X to 0}
2: $X[0] := m$; {Stores input for simulation}
3: $S := \langle 0, 0 \rangle$; {Empty stack}
4: $term := 1$;
5: $cur := g$;
6: **while** $term \neq 0$ **do**
7: $type := \pi_1(cur)$;
8: **if** $type = 0$ **then**
9: simulate addition (as in Program 10).
10: **fi**
11: **if** $type = 1$ **then**
12: simulate subtraction.
13: **fi**
14: **if** $type = 2$ **then**
15: simulate initialization with constant.
16: **fi**
17: **if** $type = 3$ **then**
18: $i := \pi_1(\pi_2(cur))$;
19: **if** $X[i] \neq 0$ **then**
20: push(S, cur); {push the loop once again}
21: push$(S, \pi_2(\pi_2(cur)))$ {push the body of the loop}
22: **fi**
23: **fi**
24: **if** $type = 4$ **then**
25: push$(S, \pi_2(\pi_2(cur)))$; {push two parts onto stack in reverse order}
26: push$(S, \pi_1(\pi_2(cur)))$
27: **fi**
28: **if** isempty$(S) = 0$ **then**
29: $cur :=$ top(S); pop(S);
30: **else**
31: $term := 0$
32: **fi**
33: **od**;
34: $x_0 := X[0]$;

The key in proving the correctness of the universal WHILE program is to show the following lemma. The subsequent Theorem 11.2 is a special case of it.

Lemma 11.1 *Let T be the state that corresponds to the content of X, let σ be the content of stack S, and let $P = \text{göd}^{-1}(cur)$ before line 6 at the beginning of the while loop. Let T' be the state that corresponds to the content of X, when the content of S after line 28 at the end of the while loop is again σ for the first time. Then $T' = \Phi_P(T)$, if this event occurs. If σ is never again the content of S, i.e., U does not terminate, then $\Phi_P(T)$ is not defined, i.e., P does not terminate when started in state T.*

Exercise 11.3 *Prove Lemma 11.1. You can use—ta dah!—structural induction.*

Theorem 11.2 *There is a WHILE program U that given $g \in \mathbb{N}$ and $m \in \mathbb{N}$, computes $\varphi_{\text{göd}^{-1}(g)}(m)$ if $\varphi_{\text{göd}^{-1}(g)}(m)$ is defined and does not terminate otherwise.*

Corollary 11.3 (Kleene normal form [Kle43]) *Let f be a WHILE computable function. Then there are FOR programs P_1, P_2, and P_3 such that the program*

$$P_1;\ \textbf{\textit{while}}\ x_1 \neq 0\ \textbf{\textit{do}}\ P_2\ \textbf{\textit{od}};\ P_3$$

computes f.[2]

Proof. Let P be some WHILE program for f and let $g = \text{göd}(P)$. Our universal program U is in Kleene normal form. (Recall that arrays, stacks, and the projections all could be realized by FOR programs.) Instead of giving U the Gödel number g as an input, we hardwire it into the program, i.e, in line 5, g is now a constant and not a variable. ∎

The above theorem tells us that in principle, one while loop is as powerful as any number of while loops. We need some for loops, though, in the simulation of the stack and the array, for instance. However, this number of for loops is also fixed!

11.1 Exercises

Basic exercises

Exercise 11.4 *Show that the following function is FOR computable:*

$$\langle i, \langle x, t\rangle\rangle \mapsto \begin{cases} 1 & \text{if program } \text{göd}^{-1}(i) \text{ halts on } x \text{ after} \leq t \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

[2] Formally, we did not define the semantics of mixed WHILE and FOR programs. But I am sure you can do it on your own.

"After $\leq t$ steps" here means after the execution of $\leq t$ simple statements. Note that by definition, the body of a while loop is never empty.

Intermediate exercises

Exercise 11.5 *Is there a universal FOR program, i.e., a FOR program U_0 that given a Gödel number i of a FOR program P and an input x computes $\varphi_P(x)$?*

12 The halting problem

The *halting problem* H is the following problem:

$$H = \{\langle g, m\rangle \mid \text{göd}^{-1}(g) \text{ halts on } m\}.$$

This is a natural problem. The *special halting problem* H_0 is the following special case:

$$H_0 = \{g \mid \text{göd}^{-1}(g) \text{ halts on } g\}.$$

Here we want to know whether a WHILE program halts on its own Gödel number. While this is not as natural as the regular halting problem, it is a little easier to prove that it is not decidable. In the next chapter, we formally show that it is indeed a special case of the halting problem and develop a general method to show that problems are not decidable.

12.1 The halting problem is not decidable

Theorem 12.1 *$H_0 \notin \mathsf{REC}$, i.e., the special halting problem is not decidable.*

Proof. The proof is by contradiction. Assume that there is a WHILE program P_0 that decides H_0, i.e., $\varphi_{P_0} = \chi_{H_0}$. In particular, P_0 always terminates. Consider the following program P:

```
1: P_0;
2: if x_0 = 1 then
3:    x_1 := 1;
4:    while x_1 ≠ 0 do
5:       x_1 := 1
6:    od
7: fi
```

What does P do? It first runs P_0. If P_0 returns 0, i.e., $x_0 = 0$ after running P_0, then P will terminate. If P_0 returns 1, then P enters an infinite loop and does not terminate. (Note that P_0 either returns 0 or 1.)

Now assume that P terminates on input göd(P). In this case, P_0 returns 0 on göd(P). But this means, that P does not terminate on göd(P), a contradiction.

If P does not terminates on göd(P), then P_0 returns 1 on göd(P). But this means that P terminates on göd(P), again a contradiction.

Since P either terminates on göd(P) or does not, this case distinction is exhaustive, and therefore, P_0 cannot exist. ∎

Excursus: Alan M. Turing and the halting problem

Alan M. Turing (born 1912, died 1954) was a British mathematician and cryptographer. He is one of the parents of Theoretical Computer Science.

He studied the halting problem (for Turing machines, a model that we will see soon and that is equivalent to WHILE programs) to show that Hilbert's Entscheidungsproblem is not decidable in 1936. (This was done indepently by Alonzo Church whom we will meet later.) The *Entscheidungsproblem* is the problem in symbolic logic to decide whether a given first-order statement is universally valid or not. It was posed in this form by D. Hilbert in 1928 in the context of *Hilbert's program*.

During World War II, Turing contributed to the British efforts of breaking the German ciphers. He died like snow white.

Further reading:
– The Turing Archive. `www.turingarchive.org`
– Andrew Hodges. *Alan Turing: The Enigma*, Walker Publishing Company, 2000.

Excursus: Programming systems III

The halting problem is not decidable for any acceptable programming system. For every acceptable programming system $(\psi_i)_{i\in\mathbb{N}}$, the special halting problem is the set $P = \{i \mid \psi_i(i) \text{ is defined}\}$. We have to show that there is no j such that $\chi_P = \psi_j$. Assume on the contrary that such an index j exists.

Consider a function f such that $f(1)$ is undefined and $f(0) = 0$. (The remaining values do not matter, we can assume that they are 0.) The function f is certainly WHILE computable, hence there is an index k such that $f = \psi_k$.

Let $\ell = c(j, k)$ where c is the composition function of the programming system. Then $\psi_\ell(\ell) = f \circ \chi_P(\ell)$. This is a contradiction, since for all x, $f \circ \chi_P(x)$ is defined if $\psi_x(x)$ is undefined and undefined if $\psi_x(x)$ is defined.

12.2 Recursively enumerable languages

While we cannot algorithmically decide whether a WHILE program P halts on an input m or not, we can at least detect if P halts on m, just by simulating P. Such a behaviour is sometimes called "semi-decidable", since we can output a 1 on all inputs $\langle g, m\rangle \in H$ (the "yes"-instances) but we cannot output 0 on the inputs $\langle g, m\rangle \notin H$ (the "no"-instances). Instead of semi-decidable, we will use the term *recursively enumerable*.

Definition 12.2 *1. A language $L \subseteq \mathbb{N}$ is called* recursively enumerable *if there is a WHILE program P such that*

(a) for all $x \in L$, $\varphi_P(x) = 1$ and

(b) for all $x \notin L$, $\varphi_P(x) = 0$ or $\varphi_P(x)$ is undefined.

2. *The set of all recursively enumerable languages is denoted by* RE.

Remark 12.3 *In condition 1.(b) of the definition above, we can always assume that $\varphi_P(x)$ is undefined. We can modify P in such a way that whenever it returns 0, then it enters an infinite loop. Thus on $x \in L$, P halts (and outputs 1), on $x \notin L$, P does not halt. We can define a "modified" characteristic function χ'_L by*

$$\chi'_L(x) = \begin{cases} 1 & \text{if } x \in L, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Then $L \in \mathsf{RE}$ iff χ'_L is WHILE computable.

Theorem 12.4 *The halting problem and the special halting problem are recursively enumerable.*

Proof. Let $\langle g, m \rangle$ be the given input. We simulate $\text{göd}^{-1}(g)$ on m using the univeral WHILE program U. It is easy to see that this program terminates if and only if $\text{göd}^{-1}(g)$ halts on m. If it terminates, then we return 1. ∎

Remark 12.5 *The set that corresponds to the characteristic function c constructed in the alternative proof of Corollary 10.2 is not recursively enumerable, since we diagonalized against all WHILE programs and not only those that compute total functions.*

Theorem 12.6 *The following two statements are equivalent:*

1. $L \in \mathsf{REC}$.
2. $L, \bar{L} \in \mathsf{RE}$.

Proof. For the "⇒" direction, note that $L \in \mathsf{REC}$ implies $\bar{L} \in \mathsf{REC}$ (c.f. Exercise 12.2) and that $\mathsf{REC} \subseteq \mathsf{RE}$.

For the other direction, note that there are WHILE programs P and $\bar{P}$ that halt on all $m \in L$ and $m \notin L$. So either P or $\bar{P}$ halts on a given m. The problem is that we do not know which. If we run P first then it might not halt on $m \in \bar{L}$ and we never have a chance to run $\bar{P}$ on m.

The trick is to run P and $\bar{P}$ in parallel. To achieve this, we modify our universal WHILE program U. In the while loop of U, we will simulate one step of P and one step of $\bar{P}$. (We need two stacks S_1, S_2 to do this, two instances cur_1, cur_2 of the variable cur, etc.) Eventually, one of the programs P or $\bar{P}$ will halt. Then we know whether $m \in L$ or not. ∎

Corollary 12.7 $\bar{H}_0 \notin \mathsf{RE}$.

Proof. We know that H_0 is in RE but not in REC. By Theorem 12.6, $\bar{H}_0$ is not in RE. ■

The following exercise explains the name "recursively enumerable":

Exercise 12.1 *Show that the following three statements are equivalent:*[1]

1. $L \in \mathsf{RE}$.
2. *There is a WHILE program* P *such that* $L = \operatorname{im} \varphi_P$.
3. $L = \emptyset$ *or there is a FOR program* P *such that* $L = \operatorname{im} \varphi_P$.

The good, the bad, and the ugly

For a language A, exactly one of the following is true:

- $A, \bar{A} \in \mathsf{REC}$
- $A \in \mathsf{RE}$ but $\bar{A} \notin \mathsf{RE}$ (or vice versa)
- $A, \bar{A} \notin \mathsf{RE}$

12.3 Exercises

Exercise 12.2 *Prove that* REC *is closed under intersection, union, and complementation, that is, if* $A, B \in \mathsf{REC}$*, so are* $A \cap B$*,* $A \cup B$*, and* $\bar{A}$*.*

Exercise 12.3 1. *Prove that* RE *is closed under union and intersection.*

2. *Prove that* RE *is* not *closed under complementation.*

[1] This explains the name recursively enumerable: There is a WHILE computable function, here φ_P, that enumerates L, that means, if we compute $\varphi_P(0), \varphi_P(1), \varphi_P(2), \ldots$, we eventually enumerate all elements of L.

13 Reductions

Let us come back to the verification problem: Does a given program match a certain specification? One very general approach to model this is the following: Given two encodings $i, j \in \operatorname{im} \text{göd}$, do the WHILE programs $\text{göd}^{-1}(i)$ and $\text{göd}^{-1}(j)$ compute the same function, i.e., is $\varphi_{\text{göd}^{-1}(i)} = \varphi_{\text{göd}^{-1}(j)}$. The index i is the program that we want to verify, the index j is the specification that it has to match. So let

$$V = \{\langle i, j\rangle \mid \varphi_{\text{göd}^{-1}(i)} = \varphi_{\text{göd}^{-1}(j)}\}.$$

One can of course complain that a WHILE program is a very powerful specification. So we will also investigate the following (somewhat artifical but undeniably simple) special case:

$$V_0 = \{i \mid \varphi_{\text{göd}^{-1}(i)}(x) = 0 \text{ for all } x \in \mathbb{N}\}.$$

So $i \in V_0$ means that the WHILE program $\text{göd}^{-1}(i)$ outputs 0 on every input (and in particular halts on every input).

Another relevant and basic problem is the termination problem: Does a WHILE program halt on *every* input:

$$T = \{i \mid \varphi_{\text{göd}^{-1}(i)} \text{ is total}\}.$$

We will prove that none of these languages is decidable, i.e, $V, V_0, T \notin \mathsf{REC}$. So there is no hope for a *general* and *automated* way to predict/prove semantic properties of WHILE programs and henceforth of computer programs in general.

We use a concept called *reductions*, a prominent concept in computer science. Assume someone gives you a JAVA procedure that finds the minimum in a list of integers. You can use this as a subroutine to quickly write a JAVA program—though not the most efficient one—that sorts a list. We will use the same principle but "in the reverse direction". Assume there is a WHILE program that decides V_0, say. Using this WHILE program, we will construct a new WHILE program that decides the halting problem H_0. Since the later does not exists, there former cannot exists either. Thus V_0 is not in REC, too.

13.1 Many-one reductions

Definition 13.1 *Let $L, L' \subseteq \mathbb{N}$ be two languages.*

1. *A WHILE computable total function $f : \mathbb{N} \to \mathbb{N}$ is called a* many-one reduction *from L to L' if*

$$\text{for all } x \in \mathbb{N}: \quad x \in L \iff f(x) \in L'.$$

2. *If such an f exists, then we say that L is* recursively many-one reducible *to L'. We write $L \leq L'$ in this case.*

Example 13.2 *Here is an example of a very simple reduction, we want to reduce $H_0 \leq H$. The reduction is given by $g \mapsto \langle g, g\rangle$, we have*

$$g \in H_0 \iff \text{göd}^{-1}(g) \text{ halts on } g \iff \langle g, g\rangle \in H.$$

The mapping is obviously WHILE computable. This is of course a very simple example. We will see more complicated examples soon. As a little brain teaser: Can you prove the other direction, that is, $H \leq H_0$?

The next lemma shows the importance of reductions. If $L \leq L'$, then the fact that L' is easy (that means, is in RE or REC) implies that L is easy, too. The contraposition of this is: If L is hard (that is, is not in RE or REC) then L' is also hard.

Lemma 13.3 *Let $L, L' \subseteq \mathbb{N}$ and $L \leq L'$. Then the following statements hold:*

1. *If $L' \in \mathsf{RE}$, then $L \in \mathsf{RE}$.*
2. *If $L' \in \mathsf{REC}$, then $L \in \mathsf{REC}$.*

Proof. Let f be a many one reduction from L to L'. We prove the first statement. Since $L' \in \mathsf{RE}$, there is a program P' such that $\varphi_{P'}(m) = 1$ for all $m \in L'$ and $\varphi_{P'}(m)$ is undefined for all $m \notin L'$. Consider the following program P:

```
1: x_0 := f(x_0);
2: P'
```

It computes $\varphi_P = \varphi_{P'} \circ f$. We have

$$x \in L \Rightarrow f(x) \in L' \Rightarrow \varphi_{P'}(f(x)) = 1 \Rightarrow \varphi_P(x) = 1$$

and

$$x \notin L \Rightarrow f(x) \notin L' \Rightarrow \varphi_{P'}(f(x)) \text{ is undefined} \Rightarrow \varphi_P(x) \text{ is undefined.}$$

Thus $L \in \mathsf{RE}$ (as witnessed by P). The second statement is shown in the same fashion. In this case, the program P' outputs 1 if $m \in L$. If $m \notin L'$, then $\varphi_{P'}(m) = 0$ instead of being undefined. Thus we only have to replace "is undefined" in the second series of implications by "$= 0$" and then the same proof works. ∎

Corollary 13.4 *Let $L, L' \subseteq \mathbb{N}$ and $L \leq L'$. Then the following statements hold:*

1. *If $L \notin \mathsf{RE}$, then $L' \notin \mathsf{RE}$.*
2. *If $L \notin \mathsf{REC}$, then $L' \notin \mathsf{REC}$.*

Proof. These statement are the contrapositions of the statements of the previous lemma. ∎

Remark 13.5 *The proof of the above lemma essentially shows that when $L \leq L'$ by a reduction f, then*

$$\chi_L = \chi_{L'} \circ f \qquad \text{and}$$
$$\chi'_L = \chi'_{L'} \circ f$$

Since f is WHILE computable and the composition of two WHILE computable functions is again WHILE computable, χ_L and χ'_L are WHILE computable, if $\chi_{L'}$ and $\chi'_{L'}$ are.

Where is the subroutine idea gone...?

Assume there is a many one reduction f from H_0 to V_0. From a (hypothetical) WHILE program P that decides V_0, we get a program Q for H_0 as follows:

```
1: x_0 := f(x_0);
2: P
```

So a many-one reduction is a very special kind of subroutine use: We only use the subroutine (P) once and the output of our program (Q) has to be the output of the subroutine.
Why do we use many-one reductions? Because it is sufficient for our needs and because we get finer results. For instance the first statement of Lemma 13.3 is not true for *Turing reductions* (which is a fancy name for arbitrary subroutine use).

Lemma 13.6 *$\leq$ is a transitive relation.*

Proof. Assume that $L \leq L'$ and $L' \leq L''$. Let f and g be the corresponding reductions. Since f and g are WHILE computable, $g \circ f$ is, too. If P and Q compute f and g, respectively, then $[P; Q]$ computes $g \circ f$.

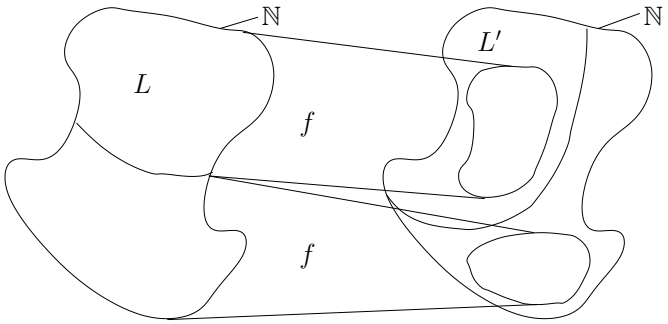


Figure 13.1: A reduction f maps the elements of L (also called "yes"-instances) to a subset of the elements of L'. The elements not in L ("no"-instances) are mapped to a subset of the elements of $\mathbb{N}$ that are not in L'. In short, "yes"-instances go to "yes"-instances and "no"-instances to "no"-instances.

We claim that $g \circ f$ is a many-one reduction from L to L'. We have

$$x \in L \iff f(x) \in L' \iff g(f(x)) \in L''$$

for all $x \in \mathbb{N}$ by the definition of many-one reduction. This completes the proof. ■

Exercise 13.1 *Show the following: If L is many-one reducible to L', then $\bar{L}$ is many-one reducible to $\bar{L}'$. (Hint: Just have a look at Figure 13.1.)*

13.2 Termination and Verification

We show that neither V_0 nor V nor T are decidable. Even worse, all three problems are not recursively enumerable.

Lemma 13.7 $H_0 \leq V_0$.

Proof. For a given input i, consider the following WHILE program Q_i, where $P := \text{göd}^{-1}(i)$:

1: $x_0 := i$;
2: P;
3: $x_0 := 0$;

Note that Q_i completely ignores its input. If P halts on its own Gödel number i, then Q_i always outputs 0, i.e., $\varphi_{Q_i}(x) = 0$ for all x. If P does not

halt on i, then Q_i never halts, that is, φ_{Q_i} is the function that is nowhere defined. In other words,

$$\text{göd}(P) \in H_0 \iff \text{göd}(Q_i) \in V_0.$$

Thus the mapping f that maps $i \mapsto \text{göd}(Q_i)$ is the desired reduction.

We only have to convince ourselves that f is indeed WHILE computable. The Gödel numbers of the three parts of Q_i:

- $\langle 2, \langle 0, i\rangle\rangle_5$.
- i
- $\langle 2, \langle 0, 0\rangle\rangle_5$.

Therefore, the reductions is given by

$$i \mapsto \langle 4, \langle\langle 2, \langle 0, i\rangle\rangle_5, \langle 4, \langle i, \langle 2, \langle 0, 0\rangle\rangle_5\rangle\rangle_5\rangle\rangle_5 \tag{13.1}$$

The above mapping is WHILE computable since the pairing functions are. So the reduction $H_0 \leq V_0$ has an easy explicit description.[1] ■

Since V_0 is a special case of V, V_0 can be reduced to V.

Lemma 13.8 $V_0 \leq V$.

Proof. The reduction is given by $i \mapsto \langle i, e\rangle$ where e is the Gödel number of a WHILE program that computes the constant function 0. ■

Verification is even harder than the halting problem, since $\bar{V}_0$ is not recursively enumerable.

Lemma 13.9 $V_0 \leq T$.

Proof. For a given $i \in \mathbb{N}$, construct the following WHILE program J_i:

```
1: P;
2: while x_0 ≠ 0 do
3:     x_0 := 1
4: od
```

If P does not compute the function that is constantly 0, then either P does not halt on some input or halts and outputs something else. In the first case, J_i does not terminate, because P does not terminate either. In the second case, J_i does not terminate, because it enters an infinite loop in lines

[1] Of course, we will see more complicated reductions for which such a formal proof would get really lengthy. We do some of the reductions, like this one, in a very explicit way, later, we will only sketch how a WHILE program for such a reduction would look like.

2—4. (Recall that the output of P_0 is stored in x_0. We need line 3, because the body of a WHILE loop cannot be empty.) Thus, J_i terminates on all inputs if and only if P outputs 0 on every input. Hence, the mapping that maps $i \mapsto \text{göd}(J_i)$ is a many-one reduction from V_0 to T. This mapping is easily seen to be WHILE computable. ■

Exercise 13.2 *Write down an explicit expression for the reduction in the previous lemma to show that it is WHILE computable.*

Lemma 13.10 $\bar{H}_0 \leq V_0$.

Proof overview: When we showed $H_0 \leq V_0$, we constructed a WHILE program Q_i that simulated $P := \text{göd}^{-1}(i)$ on i and outputted 0 if P halted. Now the program Q_i would have to output 0 if P does not halt. This straight forward approach obviously does not work here.

A parameter that we did not use are the inputs of Q_i. We will simulate P only for a finite number of steps—the larger the input, the longer the simulation.

Proof. For a given $i \in \mathbb{N}$, consider the following WHILE program K_i (using syntactic sugar at a maximum level):

1: Simulate $P := \text{göd}^{-1}(i)$ on i for x_0 steps.
2: If P does not stop within x_0 steps, then output 0.
3: Otherwise output 1.

If $i \notin H_0$, then in step 2 of K_i, P does not halt on i for any value of x_0. Thus K_i always outputs 0. If $i \in H_0$, then there is a $t \in \mathbb{N}$ such that M halts within t steps on i. Thus K_i will output 1 for every input $\geq t$.

Thus the mapping $i \mapsto \text{göd}(K_i)$ is a many-one reduction from $\bar{H}_0$ to V_0. Like before, this mapping is WHILE computable. However, this time the actual construction is much more tedious. We will not do it here. Convince yourself that you could write a WHILE program for this mapping. In the next chapter, we will see a more elegant way to design reductions. ■

Summarizing the result of this section, we get the following reductions:

$$H_0,\ \bar{H}_0 \leq V_0 \leq V,\ T.$$

For the complements of the mentioned languages, note that by Exercise 13.1, we also get

$$\bar{H}_0,\ H_0 \leq \bar{V}_0 \leq \bar{V},\ \bar{T}.$$

Thus we obtain the following theorem.

Theorem 13.11 *V, V_0, and T are not recursively enumerable nor are their complements.*

How to construct a many-one reduction ...

To reduce L to L':

1. First try to find for every $i \in L$ an $f(i)$ that is in L' and for every $i \notin L$, find an $f(i)$ that is not in L'.
   If L is defined in terms of properties of functions computed by WHILE programs (this is the "normal" case), try to map a given encoding i that has (not) the property of L to an encoding $f(i)$ that has (not) the property of L'.
2. Give a formal proof that your mapping f has indeed the reduction property.
3. "Prove" that f is WHILE computable. (A formal proof of this might be tedious. A quick argument is usually sufficient.)

14 More on reductions

From now on, for $i \in \mathbb{N}$, we will use φ_i as a synonym for $\varphi_{\text{göd}^{-1}(i)}$; this is only done to simplify notations as a reward for working through 14 chapters so far.

14.1 S-m-n Theorem

Theorem 14.1 (S-m-n Theorem) *For every $m, n \geq 1$, there is a FOR computable function $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ such that for all $g \in \mathbb{N}$, $y \in \mathbb{N}^m$, and $z \in \mathbb{N}^n$*

$$\varphi_g^{m+n}(y, z) = \varphi^n_{S_n^m(g,y)}(z). \ ^1$$

Proof overview: The statement of the theorem looks complicated at a first glance, but it just states the following simple thing: Given a Gödel number of a program that expects $n + m$ inputs and a number $y \in \mathbb{N}^m$, we can compute the Gödel number of a program that specializes the first m inputs to y. While the statement of the theorem is quite simple, it is often very useful.

Proof. Let $P_g = \text{göd}^{-1}(g)$ the program that is given by g. P_g expects $m + n$ inputs. Given a $y = (\eta_0, \ldots, \eta_{m-1})$, we now have to construct a program $Q_{g,y}$ that depends on g and y and fulfills

$$\varphi_{P_g}(y, z) = \varphi_{Q_{g,y}}(z) \quad \text{for all } z \in \mathbb{N}^n.$$

The following program $Q_{g,y}$ achieves this:

1: $x_{m+n-1} := x_{n-1}$;
2: $\vdots$
3: $x_m := x_0$;
4: $x_{m-1} := \eta_{m-1}$;

[1] The superscripts $n + m$ and n indicate the number of inputs to the program. Note that the same program can have any number of inputs since we made the convention that the inputs stand in the first variables at the beginning. So far, this number was always clear from the context. Since it is important here, we added the extra superscript.
Strictly speaking φ_g^{m+n} expects one vector of length $m+n$. y is a vector of length m and z is a vector of length n. If we glue them together, we get a vector of length $m + n$. For the sake of simplicity, we write $\varphi_g^{m+n}(y, z)$ instead of formally forming a vector of length $m + n$ out of y and z and then plugging this vector into φ_g^{m+n}.

5: $\vdots$
6: $x_0 := \eta_0$;
7: P_g

This program first copies the input z, which stands in the variables $x_0, \ldots, x_{n-1}$ into the variables $x_m, \ldots, x_{m+n-1}$. Then it stores y into $x_0, \ldots, x_{m-1}$. The values of y are hardwired into $Q_{g,y}$. Then we run P_g on the input (x, y). Thus $Q_{g,y}$ computes $\varphi_{P_g}(y, z)$ but only the entries from z are considered as inputs.

The function

$$S_n^m : (g, y) \mapsto \text{göd}(Q_{g,y})$$

is FOR computable. We saw how to show this in the last chapter. However, it is also fairly easy to write down a Gödel number explicity, which we will do now (for the last time): Note that the lines 1 to 3 of $Q_{g,y}$ do not depend on g or y (but only on m and n). Therefore, this part has some fixed Gödel number, say, h. The lines 4 to 6 depend on y and have Gödel number

$$h' = \langle 4, \langle\langle 2, \langle m-1, \eta_{m-1}\rangle\rangle_5, \langle 4, \langle\langle 2, \langle m-1, \eta_{m-1}\rangle\rangle_5, \ldots\rangle\rangle_5, \rangle\rangle_5.$$

As a function of y, h' is FOR computable, since the pairing functions are. Thus $\text{göd}(Q_{g,y}) = \langle 4, \langle h, \langle 4, \langle h', g\rangle\rangle_5\rangle\rangle_5$.

This mapping above has the desired properties, since

$$\varphi_g^{m+n}(y, z) = \varphi_{P_g}^{m+n}(y, z) = \varphi_{Q_{g,y}}^n(z) = \varphi_{\text{göd}(Q_{g,y})}^n(z) = \varphi_{S_n^m(g,y)}^n(z)$$

for all $y \in \mathbb{N}^m$ and $z \in \mathbb{N}^n$. ■

Excursus: Programming systems IV

The S-m-n theorem is valid for every acceptable programming system (ψ_i), but a little more work is needed. We will here give a proof for the case $m = n = 1$, the general case can be reduced to this by using pairing functions. Programming systems only compute functions $\mathbb{N} \to \mathbb{N}$. The function $\psi_i^k : \mathbb{N}^k \to \mathbb{N}$ is now defined as $\psi_i^k(x_1, \ldots, x_k) = \psi_i(\langle x_1, \ldots, x_k\rangle)$.

Let $f(z) = \langle 0, z\rangle$ and $g(y, z) = \langle y + 1, z\rangle$. Both functions are WHILE computable, hence there are indices i and j such that $f = \psi_i$ and $g = \psi_j^2$. Finally, we define a function h by

$$\begin{aligned} h(0) &= i, \\ h(x+1) &= c(j, h(x)) \quad \text{for } x > 0. \end{aligned}$$

where c is the composition function. h is certainly WHILE computable; we can just start with $h(0) = i$, then compute $h(1)$, $h(2)$, $h(3)$,... by applying c until we get to the desired value.

Lemma 14.2 $\psi_{h(y)}(z) = \langle y, z\rangle$.

Proof. The proof is by induction.
Induction base: $\psi_{h(0)}(z) = \psi_i(z) = \langle 0, z\rangle$.
Induction step: Assume that $\psi_{h(y)}(z) = \langle y, z\rangle$. Now,

$$\begin{aligned}\psi_{h(y+1)}(z) &= \psi_{c(j,h(y))}(z)\\ &= \psi_j \circ \psi_{h(y)}(z)\\ &= \psi_j(\langle y, z\rangle)\\ &= \langle y+1, z\rangle. \qquad \blacksquare\end{aligned}$$

We set $S_1^1(e, y) = c(e, h(y))$. We have

$$\begin{aligned}\psi_{S_1^1(e,y)}(z) &= \psi_{c(e,h(y))}(z)\\ &= \psi_e \circ \psi_{h(y)}(z)\\ &= \psi_e^2(y, z).\end{aligned}$$

The S-m-n Theorem will be the key to the results of this and the next chapter.

14.2 Reductions via the S-m-n Theorem

The S-m-n Theorem can be used to prove that a language is reducible to another one. We here give an alternative proof of $\bar{H}_0 \leq V_0$.

Alternative proof of Lemma 13.10. Consider the function $f : \mathbb{N}^2 \to \mathbb{N}$ defined by

$$f(g, m) = \begin{cases} 0 & \text{if } \text{göd}^{-1}(g) \text{ does not halt on } g \text{ after} \leq m \text{ steps} \\ 1 & \text{otherwise} \end{cases}$$

The function f is WHILE computable, we can use the clocked version of the universal WHILE program U from Exercise 11.4. Let e be a Gödel number of f. By the S-m-n Theorem,

$$f(g, m) = \varphi_e^2(g, m) = \varphi_{S_1^1(e,g)}(m)$$

for all $g, m \in \mathbb{N}$. But by construction,

$$\begin{aligned} g \in \bar{H}_0 &\iff \text{göd}^{-1}(g) \text{ does not halt on } g\\ &\iff f(g, m) = 0 \text{ for all } m \in \mathbb{N}\\ &\iff S_1^1(e, g) \in V_0.\end{aligned}$$

Thus $S_1^1(e, .)$ is the desired many one reduction. $\blacksquare$

14.3 More problems

Here is another example, more of pedagogical value. Let $c \in \mathbb{N}$. Let

$$D_c = \{g \mid g \in \mathbb{N} \text{ and } |\operatorname{dom} \varphi_g| \geq c\}$$

be the set of all encodings of WHILE programs that compute a function that is defined for at least c different arguments. Here is potential application: As the last assignment of your programming lecture, you have to deliver a program. You still need one point to be qualified for the exam. The TA claims that your program does not halt on any input and you get no points for your program. We will show that $D_1 \notin \mathsf{REC}$. This is good for you, since it means that the TA will not be able to algorithmically verify his claim. On the other hand, we will show that $D_1 \in \mathsf{RE}$, which is again good for you, since it means that if your program halts on at least one input, you can algorithmically find this input and maybe get the missing point...

Theorem 14.3 *For every $c \geq 1$, $H_0 \leq D_c$.*

Proof. Consider the following function $f : \mathbb{N}^2 \to \mathbb{N}$ defined by

$$f(i,x) = \begin{cases} 0 & \text{if } \varphi_i(i) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

f is WHILE computable: Since $f(i,x)$ is undefined if $\varphi_i(i)$ is undefined, we can just simulate $\text{göd}^{-1}(i)$ on i and return 0 if $\text{göd}^{-1}(i)$ halts.

Since f is WHILE computable, there is an index $e \in \mathbb{N}$ such that $f = \varphi_e$. By the S-m-n Theorem,

$$f(i,x) = \varphi_e^2(i,x) = \varphi_{S_1^1(e,i)}(x) \qquad \text{for all } i \text{ and } x.$$

Let $i \in H_0$. Then $f(i,x)$ is defined for all x by construction, i.e., the function $x \mapsto f(i,x)$ is total and in particular, its domain has at least c elements. Thus $S_1^1(e,i) \in D_c$. Let $i \notin H_0$. Then $f(i,x)$ is not defined for all x by construction. Thus $S_1^1(e,i) \notin D_c$. The function $i \mapsto S_1^1(e,i)$ is recursive by the S-m-n theorem (note that e is just a fixed number), thus it is the desired reduction. ∎

Theorem 14.4 *For every c, $D_c \in \mathsf{RE}$.*

Proof. We prove that χ'_{D_c} is WHILE-computable. Consider the following WHILE program P:

Input: $g \in \mathbb{N}$

1: $m := 0$;
2: $t := 1$;

```
3: while t ≠ 0 do
4:    Interpret m as a tuple (x_1, ..., x_c, i)
5:    if x_1, ..., x_c are pairwise distinct then
6:       simulate g on x_1, ..., x_c for i steps one after another
7:       if all c simulations halt then
8:          t := 0
9:       fi
10:   fi
11:   m := m + 1;
12: od
13: return 1;
```

When we say that we interpret m as a tuple $(x_1, \ldots, x_c, i)$, we mean that we use a bijection between $\mathbb{N}$ and $\mathbb{N}^{c+1}$. In this way, by increasing m every time in the loop, we will systematically enumerate all tuples in $\mathbb{N}^{c+1}$.

If P halts and outputs 1, then it necessarily set $t := 0$ in line 8. In this case, it found c pairwise distinct numbers such that g halted on all. Such $\operatorname{dom}|\varphi_g| \geq c$. On the other hand, if $\operatorname{dom}|\varphi_g| \geq c$ then let $\xi_1, \ldots, \xi_c \in \operatorname{dom}\varphi_g$ be pairwise distinct. Since ξ_i is in $\operatorname{dom}\varphi_g$, g will halt on ξ_i for $1 \leq i \leq c$. Let τ be the maximum number of steps made by g on $\xi_1, \ldots, \xi_c$. Then, when the value of m corresponds to $(\xi_1, \ldots, \xi_c, \tau)$ in the while loop, all simulations will stop and P will exit the loop. Therefore, $\varphi_P = \chi'_{D_c}$. ■

The golden rule of confusion

If something is not complicated enough,
invent many names for the same thing.
(In our case, history is responsible for this)

For a language $L \subseteq \mathbb{N}$, the following statements are equivalent or just mean the same thing by definition:
$L \in \mathsf{REC}$, L is decidable, L is recursive, the characteristic function χ_L is WHILE computable.

Also the following statements are equivalent or mean the same: $L \in \mathsf{RE}$, L is recursively enumerable, χ'_L is WHILE computable (**do not overlook the prime!**).

For a function $f : \mathbb{N} \to \mathbb{N}$, the following statements are equivalent or mean the same: f is recursive (**but f can be partial!**) and f is WHILE computable.

15 Rice's Theorem

We saw numerous proofs that certain languages are not decidable. Rice's Theorem states that any language L is not decidable if it is defined in semantic terms. This means that whether $i \in \mathbb{N}$ is in L only depends on φ_i, the function computed by the program $\text{göd}^{-1}(i)$.

15.1 Recursion Theorem

Theorem 15.1 (Recursion Theorem) *For every WHILE computable function $f : \mathbb{N}^{n+1} \to \mathbb{N}$, there is a $g \in \mathbb{N}$ such that*

$$\varphi_g^n(z) = f(g, z) \qquad \text{for all } z \in \mathbb{N}^n.$$

Proof overview: Let $f(g, z) = \varphi_e(g, z)$. Now the S-m-n Theorem states that $f(g, z) = \varphi_{S_n^1(e,g)}(z)$ for all z. If we now set $g = e$, then we are almost there: we have e on the left-hand side and $S_n^1(e, e)$ on the right-hand side which is "almost the same". If we now replace g, the first argument of f, by something of the form $S_n^1(y, y)$, then basically the same argument gives the desired result.

Proof. The function h defined by

$$h(y, z) := f(S_n^1(y, y), z) \qquad \text{for all } y \in \mathbb{N},\ z \in \mathbb{N}^n$$

is WHILE computable. Let e be a Gödel number for h, i.e., $h = \varphi_e$. The S-m-n Theorem implies that

$$\varphi_e^{n+1}(y, z) = \varphi_{S_n^1(e,y)}^n(z) \qquad \text{for all } z \in \mathbb{N}^n.$$

If we now set $y = e$ and $g = S_n^1(e, e)$, we get

$$f(g, z) = f(S_n^1(e, e), z) = h(e, z) = \varphi_e^{n+1}(e, z) = \varphi_{S_n^1(e,e)}^n(z) = \varphi_g^n(z). \qquad \blacksquare$$

Remark 15.2 *Given the index e of h, we can compute g by a WHILE machine.*

15.1.1 Self reference

Consider the function $s : \mathbb{N}^2 \to \mathbb{N}$ given by $s(y, z) = y$ for all $y, z \in \mathbb{N}$. This function is certainly WHILE computable. By the Recursion Theorem, there is an index g such that

$$\varphi_g(z) = s(g, z) = g \qquad \text{for all } z \in \mathbb{N}.$$

This means that the WHILE program given by the Gödel number g computes the constant function with value g and in this sense outputs its own source code.

Exercise 15.1 *Show that there is a Gödel number j with* $\operatorname{dom} \varphi_j = \{j\}$.

15.1.2 The halting problem again

Here is an alternative proof that the halting problem H is not decidable: Assume that H is decidable and P is a WHILE program that decides H. Then following function

$$f(e, x) = \begin{cases} 0 & \text{if } \varphi_e(x) \text{ is undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

is WHILE computable since we can check whether $\varphi_e(x)$ is defined by envoking P. By the Recursion Theorem, there is an index e_0 such that

$$\varphi_{e_0}(x) = f(e_0, x).$$

Assume that $\varphi_{e_0}(e_0)$ is defined. Then $f(e_0, e_0) = \varphi_{e_0}(e_0)$ is undefined by construction, a contradiction. But if $\varphi_{e_0}(e_0)$ were undefined, then $f(e_0, e_0) = \varphi_{e_0}(e_0) = 0$, a contradiction again. Thus H cannot be decidable.

15.1.3 Code minimization

Finally, consider the following language

$$\text{Min} = \{g \mid \text{for all } g' \text{ with } \varphi_g = \varphi_{g'},\ g \leq g'\}.$$

This is the set of all minimal WHILE programs ("shortest source codes") in the sense that for every $g \in \text{Min}$, whenever g' computes the same functions as g, then $g \leq g'$.

Theorem 15.3 $\text{Min} \notin \mathsf{RE}$.

Proof. The proof is by contradiction. Assume that $\text{Min} \in \mathsf{RE}$. Then there is a WHILE computable total function h such that $\text{Min} = \operatorname{im} h$. The function

$$f : (g, w) \mapsto f(g, w) = \varphi_k(w) \qquad \text{with } k = h(j) \text{ and } j = \min\{i \mid g < h(i)\}$$

is WHILE computable since k can be determined by a WHILE program: We successively compute $h(0), h(1), h(2), \ldots$ until we hit a j such that $g < h(j)$. Such a j exists, since Min is infinite.

By the recursion theorem, there is a Gödel number e such that

$$\varphi_e(w) = f(e, w) = \varphi_k(w) \qquad \text{for all } w \in \mathbb{N}.$$

By the construction of f, $e < k$. But we also have $k \in \text{Min} = \operatorname{im} h$. This is a contradiction, as $\varphi_e = \varphi_k$ and $e < k$ implies $k \notin \text{Min}$. ∎

15.2 Fixed Point Theorem

A *fixed point* of a function $f : \mathbb{N} \to \mathbb{N}$ is a $z_0 \in \mathbb{N}$ such that $f(z_0) = z_0$. Not every function has a fixed point, $z \mapsto z + 1$ is an example. But every WHILE computable total function f has a *semantic* fixed point in the sense that z_0 and $f(z_0)$ are the Gödel numbers of WHILE machines that compute the same function.

Theorem 15.4 (Fixed Point Theorem) *For all WHILE computable total functions $f : \mathbb{N} \to \mathbb{N}$ and for all $n \in \mathbb{N} \setminus \{0\}$ there is an $e \in \mathbb{N}$ such that*

$$\varphi^n_{f(e)} = \varphi^n_e.$$

Proof. Let $g(z, y) = \varphi^n_{f(z)}(y)$ for all $z \in \mathbb{N}$, $y \in \mathbb{N}^n$. g is WHILE computable since f is WHILE computable and total.

By the Recursion Theorem, there is an $e \in \mathbb{N}$ such that

$$\varphi^n_e(y) = g(e, y) = \varphi^n_{f(e)}(y) \qquad \text{for all } y \in \mathbb{N}^n. \qquad ∎$$

15.3 Rice's Theorem

Definition 15.5 (Index set) *A language $I \subseteq \mathbb{N}$ is an* index set *if*

$$\textit{for all } i, j \in \mathbb{N}\textit{: } i \in I \textit{ and } \varphi_i = \varphi_j \implies j \in I.$$

An index set I is nontrivial *if, in addition, $I \neq \emptyset$ and $I \neq \mathbb{N}$.*

Above, when we write φ_i, we mean φ^1_i as usual, that is, we consider WHILE programs with one input. This is done for the sake of simplicity, the same would work for other numbers of inputs.

Exercise 15.2 *Show that I is an index set if and only if there is a set F of WHILE computable functions such that $I = \{i \in \mathbb{N} \mid \varphi_i \in F\}$.*

If an index set contains a Gödel number i, then it contains all Gödel numbers j of WHILE programs that compute the same function as $\text{göd}^{-1}(i)$. In this sense, the index sets are defined by semantic properties, i.e., properties that only depend on φ_i.

Example 15.6 *The following languages are index sets:*

1. $V_0 = \{i \in \mathbb{N} \mid \varphi_i(x) = 0 \text{ for all } x \in \mathbb{N}\}$.
2. $T = \{i \in \mathbb{N} \mid \varphi_i \text{ is total}\}$
3. $D_c = \{i \in \mathbb{N} \mid |\operatorname{dom} \varphi_i| \geq c\}$ *for any* $c \in \mathbb{N}$,
4. $\text{Mon} = \{i \in \mathbb{N} \mid \varphi_i \text{ is monotone}\}$.

All of them are nontrivial except D_0.

Example 15.7 *The following sets are not index sets:*

1. H_0, *the special halting problem,*
2. $N_1 = \{g \in \mathbb{N} \mid g \leq 10000\}$.

H_0 is not an index set, since we constructed a Gödel number j with $\operatorname{dom} \varphi_j = \{j\}$ in Exercise 15.1. Thus $j \in H_0$ but any other Gödel number k that computes the same function does not halt on its own Gödel number k.

N_1 is not an index set since for any function f, there are arbitrarily large WHILE programs, i.e., with arbitrarily large Gödel numbers, that compute f.

Theorem 15.8 (Rice's Theorem) *Every nontrivial index set is not decidable.*

Proof. Let I be a nontrivial index set. Since I is nontrivial, there are Gödel numbers i and j such thats $i \in I$ but $j \notin I$. If I were decidable, then the function $h : \mathbb{N} \to \mathbb{N}$ defined by

$$h(x) = \begin{cases} i & \text{if } x \notin I \\ j & \text{if } x \in I \end{cases}$$

would be WHILE computable. There is a Gödel number e by the Fixed Point Theorem such that

$$\varphi_e = \varphi_{h(e)}.$$

If $e \in I$, then $\varphi_e = \varphi_j$. But since I is an index set and $j \notin I$, we get $e \notin I$, a contradiction. If $e \notin I$, then $\varphi_e = \varphi_i$. But since I is an index set and $i \in I$, we get $e \in I$, a contradiction again. Thus I cannot be decidable. ∎

Rice's Theorem essentially says that *every* nontrivial semantic property of WHILE programs is undecidable!

15.4 Further exercises

There are nontrivial index sets that are recursively enumerable, for instance, D_c. Others are not, like T or V_0. Here is a criterion that is useful to prove that an index set is not in RE.

Exercise 15.3 *Let I be a recursively enumerable index set. Show that for all $g \in I$, there is an $e \in I$ with $\operatorname{dom} \varphi_e \subseteq \operatorname{dom} \varphi_g$ and $\operatorname{dom} |\varphi_e|$ is finite.*

III Gödel's incompleteness theorem

Loosely speaking, Gödel's incompleteness theorem states that there are formulas that are true but we cannot prove that they are true. Formulas here means quantified arithmetic formulas, i.e., we have formulas with existential and universal quantifiers over the natural numbers with addition and multiplication as our operations. "We cannot prove" means that there is no effective way to show that the formula is true.

III.1 Arithmetic terms and formulas

Definition III.1 *Let $V = \{x_0, x_1, x_2, \dots\}$ be a set of variables.*[1] Arithmetic terms *over V are defined inductively:*

1. *Every $n \in \mathbb{N}$ is an arithmetic term.*
2. *Every $x \in V$ is an arithmetic term.*
3. *If s and t are arithmetic term, then $(s + t)$ and $(s \cdot t)$ are arithmetic terms, too.*
   (These are words over the infinite alphabet $\mathbb{N} \cup V \cup \{(,), +, \cdot\}$, not polynomials or something like that.)

Definition III.2 Arithmetic formulas *are defined inductively:*

1. *If s and t are terms, then $(s = t)$ is an arithmetic formula.*
2. *If F and G are arithmetic formulas, then $\neg F$, $(F \vee G)$, and $(F \wedge G)$ are arithmetic formulas.*
3. *If x is a variable and F is an arithmetic formula, then $\exists x F$ and $\forall x F$ are arithmetic formulas.*

Let F and G be formulas. We define the fact that G is a *subformula* of F inductively: If $G = F$, then G is a subformula of F. If $F = \neg F_1$ and G is a subformula of F_1, then G is also a subformula of F. In the same way, if $F = (F_1 \vee F_2)$ or $F = (F_1 \wedge F_2)$ or $F = \exists F_1$ or $F = \forall x F_1$ and G is a subformula of F_1 or F_2, then G is also a subformula of F.

Let x be a variable and F be a formula. The occurences of x in F are these position in F that contain the symbol x. An occurence of x in the

[1] As usual, we will use other names for variables, too.

formula F is bounded if this occurence is contained in a subformula of F of the form $\exists xG$ or $\forall xG$. An ocurrence that is not bounded it is called *free*. $F(x/n)$ denotes the formula that we get if we replace every free occurence of x in F by $n \in \mathbb{N}$.

A mapping $a : V \to \mathbb{N}$ is called an assignment. We extend a to the set of arithmetic terms in the obvious way:

$$\begin{array}{rcll} a(n) & = & n & \text{for all } n \in \mathbb{N}, \\ a(s+t) & = & a(s)+a(t) & \text{for all terms } s \text{ and } t, \\ a(s \cdot t) & = & a(s)a(t) & \text{for all terms } s \text{ and } t \end{array}$$

Definition III.3 *We define* true formulas *inductively:*

1. *If s and t are terms, then $(s = t)$ is true if $a(s) = a(t)$ for all assignments a.*
2. *$F = \neg F_1$ is a true formula, if F_1 is not a true formula*
3. *$F = (F_1 \vee F_2)$ is a true formula if F_1 or F_2 are true formulas.*
4. *$F = (F_1 \wedge F_2)$ is a true formulas if F_1 and F_2 are true formulas.*
5. *$F = \exists xF_1$ is a true formula if there is an $n \in \mathbb{N}$ such that $F_1(x/n)$ is a true formula.*
6. *$F = \forall xF_1$ is a true formula if for all $n \in \mathbb{N}$, $F_1(x/n)$ is a true formula.*

A formula that is not true is called false.

We define a function e that is an injective mapping from the set of all arithmetic terms and formulas to $\mathbb{N}$. It is defined inductively, in the same manner we defined the mapping göd. It is not bijective, but this does not matter.

1. $e(n) = \langle 0, n\rangle$ for all $n \in \mathbb{N}$.
2. $e(x_i) = \langle 1, i\rangle$ for all $i \in \mathbb{N}$.
3. $e(s+t) = \langle 2, \langle e(s), e(t)\rangle\rangle$ for all terms s and t.
4. $e(s \cdot t) = \langle 3, \langle e(s), e(t)\rangle\rangle$ for all terms s and t.
5. $e(s = t) = \langle 4, \langle e(s), e(t)\rangle\rangle$ for all terms s and t.
6. $e(\neg F) = \langle 5, e(F)\rangle$ for all formulas F.
7. $e(F \vee G) = \langle 6, \langle e(F), e(G)\rangle\rangle$ for all formulas F and G.
8. $e(F \wedge G) = \langle 7, \langle e(F), e(G)\rangle\rangle$ for all formulas F and G.

9. $e(\exists x_i F) = \langle 8, \langle i, e(F)\rangle\rangle$ for all formulas F.
10. $e(\forall x_i F) = \langle 9, \langle i, e(F)\rangle\rangle$ for all formulas F.

It is easy to see that the set $\operatorname{im} e$ is decidable. We can use a stack to analyze whether a given number corresponds to a correctly formed formula. Like for göd, the concrete construction is not so important.

Definition III.4 *The set of all encodings of true formulas is denoted by* $\mathcal{T}$.

III.2 Computability and representability

In this section we establish a link between formulas and WHILE programs. If F is a formula and $y_1, \ldots, y_k$ are exactly these variables that occur free in in F, then we indicate this by writing $F(y_1, \ldots, y_k)$. In this context, instead of writing $F(y_1/n_1, \ldots, y_k/n_k)$, we often just write $F(n_1, \ldots, n_k)$ for the formula in which every free occurence of y_κ is replaced by n_κ, $1 \leq \kappa \leq k$.

Definition III.5 *A function* $f : \mathbb{N}^k \to \mathbb{N}$ *is called* arithmetically representable *if there is an arithmetic formula* $F(y_1, \ldots, y_k, z)$ *such that*

$$f(n_1, \ldots, n_k) = s \iff F(n_1, \ldots, n_k, s) \text{ is true}$$

for all $n_1, \ldots, n_k, s \in \mathbb{N}$.[2] *In the same way, we define arithmetical representability for funcions* $f : \mathbb{N}^k \to \mathbb{N}^m$.

Recall that partial functions $A \to B$ are merely relations $R \subseteq A \times B$ such that for every $a \in A$, there is at most one b with $(a, b) \in R$. In the definition above we want that R can be expressed by an arithmetic formula.

Example III.6

1. *The addition function is arithmetically representable by*
$$(z = (y_1 + y_2)).$$
In the same way, we can represent the multiplication function.

2. *The modified difference function* $(y_1, y_2) \mapsto \max\{y_2 - y_1, 0\}$ *is arithmetically representable by*
$$((y_1 + z) = y_2) \vee ((y_1 > y_2) \wedge (z = 0))).$$
Above, $(y_1 > y_2)$ *shorthands*
$$\exists h(y_1 = y_2 + h + 1).$$

[2] If $f(n_1, \ldots, n_k)$ is undefined, then $F(n_1, \ldots, n_k, s)$ is not true for all $s \in \mathbb{N}$.

3. *Division with remainder is arithmetically representable, too:* $y_1 \,/\, y_2$ *is represented by*

$$\exists r((r < y_2) \wedge (y_1 = z \cdot y_2 + r))$$

and $y_1 \bmod y_2$ *is represented by*

$$\exists q((r < y_2) \wedge (y_1 = q \cdot y_2 + r)).$$

III.2.1 Chinese remaindering and storing many number in one

Our goal is to show that every WHILE computable function is arithmetically representable. One ingredient in this construction is a method of storing many natural numbers in one natural number. We saw such a method when we constructed dynamic arrays in WHILE programs. To access the elements of the arrays we used a FOR loop to repeatedly apply one of the two "inverse" functions π_1 and π_2 of the pairing function $\langle .,. \rangle$. Arithmetic formulas have only a fixed number of variables, too, but do not have FOR loops. Therefore we here construct another method for storing many values in one.

Theorem III.7 (Chinese remainder theorem) *Let* $n_1, \ldots, n_t$ *be pairwise coprime, i.e.,* $\gcd(n_i, n_j) = 1$ *for* $i \neq j$*. Then the mapping*

$$\begin{aligned}\pi_{n_1,\ldots,n_t} : \{0, \ldots, n_1 \cdots n_t - 1\} &\to \{0, \ldots, n_1 - 1\} \times \cdots \times \{0, \ldots, n_t - 1\} \\ m &\mapsto (m \bmod n_1, \ldots, m \bmod n_t)\end{aligned}$$

is a bijection.[3]

Proof. The proof is by induction on t.
Induction base: Assume that $t = 2$.[4] Since n_1 and n_2 are coprime, there are integers c_1 and c_2 such that $1 = c_1 n_1 + c_2 n_2$.[5] Then

$$c_1 n_1 \bmod n_2 = 1 \quad \text{and} \quad c_2 n_2 \bmod n_1 = 1.$$

Since the sets $\{0, \ldots, n_1 n_2 - 1\}$ and $\{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\}$ have cardinality $n_1 n_2$, it is sufficient to show that the mapping is surjective. Let $(a_1, a_2) \in \{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\}$ be given. Consider $a = a_1 c_2 n_2 + a_2 c_1 n_1$. We have

$$a \bmod n_1 = a_1 \quad \text{and} \quad a \bmod n_2 = a_2$$

[3] $i \bmod j$ here denotes the unique integer $r \in \{0, 1, \ldots, j - 1\}$ such that $i = qj + r$ for some q.

[4] We could also assume that $t = 1$. Then the induction base would be trivial. But it turns out that we would have to treat the case $t = 2$ in the induction step, so we can do it right away.

[5] We can get such integers, the so-called *cofactors*, via the extended Euclidian algorithm for computing gcds.

since $1 = c_1 n_1 + c_2 n_2$. a might not be in $\{0, 1, \ldots, n_1 n_2 - 1\}$, but there is an integer i such that $a' = a + i n_1 n_2$ is. Since $n_1, n_2 | i n_1 n_2$,

$$a' \bmod n_1 = a_1 \quad \text{and} \quad a' \bmod n_2 = a_2,$$

too.
Induction step: Let $N = n_2 \cdots n_t$ with $t > 2$. n_1 and N are coprime. By the induction hypothesis, the mappings $\pi_{n_2,\ldots,n_t}$ and $\pi_{n_1,N}$ are bijections. We have $(m \bmod N) \bmod n_i = i \bmod n_i$ for all $2 \leq i \leq t$, since $n_i | N$. Thus

$$\pi_{n_1,\ldots,n_t}(m) = (m_1, \pi_{n_2,\ldots,n_t}(m_2))$$

where $\pi_{n_1,N} = (m_1, m_2)$. Since both mappings above are bijections, their "composition" $\pi_{n_1,\ldots,n_t}$ is a bijection, too. ∎

Lemma III.8 *The number $1 + i \cdot s!$, $1 \leq i \leq s$, are pairwise coprime.*

Proof. Assume there are $i < j$ and a prime number p such that $p | (1 + i \cdot s!)$ and $p | (1 + j \cdot s!)$. Thus $p | ((j - i) \cdot s!)$. Since $0 \leq j - i \leq s$ and p is prime, $p | s!$. From this $p | 1$ follows. ∎

Lemma III.9 *For all numbers $a_1, \ldots, a_k$, there are numbers A and S and a formula $M(x, u, v, w)$ such that $M(x, \kappa, A, S)$ is true if and only if we substitute a_κ for x.*

Proof. Consider

$$M(x, u, v, w) = (x = v \mod (1 + uw)) \wedge (v < 1 + uw).$$

Now set $s = \max\{a_1, \ldots, a_k, k\}$ and $S = s!$. By Lemma III.8, the numbers $1 + iS$, $1 \leq i \leq s$ are pairwise coprime. By the Chinese remainder theorem, there is an A such that

$$a_\kappa = A \mod (1 + \kappa S)$$

for $1 \leq \kappa \leq k$, since $a_\kappa \leq 1 + \kappa S$ by the definition of s and S. Thus the formula $M(a_\kappa, \kappa, A, S)$ is true. The second part of M ensures that no other value fulfills $M(x, \kappa, A, S)$. ∎

III.2.2 Main result

Theorem III.10 *Every WHILE computable function is arithmetically representable.*

Proof. We show by structural induction that for every WHILE program P then function $\Phi_P : \mathbb{N}^{\ell+1} \to \mathbb{N}^{\ell+1}$, where ℓ is the largest index of a variable in P, is arithmetically representable by a formula $F_P(y_0, \ldots, y_\ell, z_1, \ldots, z_\ell)$.[6] From this statement, the statement of the theorem follows, since

$$F(y_0, \ldots, y_s, z) = \exists a_1 \ldots \exists a_\ell F_P(y_0, \ldots, y_s, 0, \ldots, 0, z, a_1, \ldots, a_\ell)$$

represents the function $\mathbb{N}^{s+1} \to \mathbb{N}$ computed by P.
Induction base: If $P = x_i := x_j + x_k$, then we set

$$F_P(y_0, \ldots, y_\ell, z_0, \ldots, z_\ell) = (z_i = y_j + y_k) \wedge \bigwedge_{m \neq i} z_m = y_m.$$

If $P = x_i := x_j - x_k$, then F_P looks similar, we just replace the $(z_i = y_j + y_k)$ part by the formula for the modified difference from Example III.6.

If $P = x_i := c$, then we replace the $(z_i = y_j + y_k)$ part by $(z_i = c)$.
Induction step: We first consider the case $P = [P_1; P_2]$. By the induction hypothesis, the functions $\Phi_{P_i} : \mathbb{N}^{\ell+1} \to \mathbb{N}^{\ell+1}$ [7] are arithmetically representable by formulas F_{P_i}, $i = 1, 2$. We have $\Phi_P = \Phi_{P_2} \circ \Phi_{P_1}$. The formula

$$F_P = \exists a_0 \ldots \exists a_\ell F_{P_1}(y_0, \ldots, y_\ell, a_0, \ldots, a_\ell) \wedge F_{P_2}(a_0, \ldots, a_\ell, z_0, \ldots, z_\ell)$$

represents Φ_P; the variables $a_0, \ldots, a_\ell$ "connect" the two formulas in such a way that the output of Φ_{P_1} becomes the input of Φ_{P_2}.

It remains the case $P =$ **while** $x_i \neq 0$ **do** P_1 **od**. Let F_{P_1} be a formula that represents Φ_{P_1} This is more complicated, since we have to "connect" a formula F_{P_1} for an unknown number of times. We will use the formula M from Lemma III.9.

$$\begin{aligned}
F_P = {} & \exists A_0 \exists S_0 \ldots \exists A_\ell \exists S_\ell \exists t \\
& \big(M(y_0, 0, A_0, S_0) \wedge \cdots \wedge M(y_\ell, 0, A_\ell, S_\ell) \wedge \\
& M(z_0, t, A_0, S_0) \wedge \cdots \wedge M(z_\ell, t, A_\ell, S_\ell) \wedge \\
& \forall \tau \exists v ((\tau \geq t) \vee (M(v, \tau, A_i, S_i) \wedge (v > 0))) \wedge \\
& M(0, t, A_i, S_i) \wedge \\
& \forall \tau \exists a_0 \ldots \exists a_\ell \exists b_0 \ldots \exists b_\ell \\
& \big(F_{P_1}(a_0, \ldots, a_\ell, b_0, \ldots, b_\ell) \wedge \\
& M(a_0, \tau, A_0, S_0) \wedge \cdots \wedge M(a_\ell, \tau, A_\ell, S_\ell) \wedge \\
& M(b_0, \tau + 1, A_0, S_0) \wedge \cdots \wedge M(b_\ell, \tau + 1, A_\ell, S_\ell) \\
& \vee (\tau \geq t) \big) \big)
\end{aligned}$$

[6] For simplicity, Φ_P now operates on tuples instead of functions with finite support.
[7] P_1 or P_2 might not contain the variable x_ℓ. We pad Φ_{P_i} to a function $\mathbb{N}^{\ell+1} \to \mathbb{N}^{\ell+1}$ in the obvious way.

The variable t denotes the number of times the while loop is executed. The variables A_i and S_i store values that encode the values that x_i attains after each execution of P_1 in the while loop. The second line of the definition of F_P ensures that before the first execution, the value of the variable x_λ is y_λ, $0 \leq \lambda \leq \ell$. The third line ensures that after the tth execution, the value of the variable x_λ is z_λ, $0 \leq \lambda \leq \ell$. The fourth and fifth line ensure that the first time that x_i contains the value 0 is after the tth execution of the while loop. The remainder of the formula ensures that the values that $x_0, \ldots, x_\ell$ have after the $(\tau+1)$th execution of the while loop are precisely the values that we get if we run P_1 with $x_0, \ldots, x_\ell$ containing the values after τth execution. Note that the formula M is satisfied by at most one value for fixed τ, A_λ, and S_λ. This ensure consistency, i.e, even if we A_λ and S_λ do not contain the values from Lemma III.9, if the formula F_P is satisfied, then the values stored in A_λ and S_λ, $0 \leq \lambda \leq \ell$, correspond to an execution of the WHILE program P. ∎

Remark III.11 *Furthermore, there is a WHILE program that given* $\text{göd}(P)$*, computes the encoding of a formula presenting* φ_P*.*

Lemma III.12 *If* $\mathcal{T} \in \mathsf{RE}$*, then* $\mathcal{T} \in \mathsf{REC}$*.*

Proof. Let f be a total WHILE computable functions such that $\operatorname{im} f = \mathcal{T}$. A WHILE program P that decides $\mathcal{T}$ first checks whether a given input $x \in \operatorname{im} e$. Let $e(F) = x$. P successively computes $f(0), f(1), f(2), \ldots, f(i), \ldots$ until either $e(F) = f(i)$ for $e(\neg F) = f(i)$. In the first case, P outputs 1, in the second, 0. Since either F or $\neg F$ is true, P halts on all inputs and therefore decides $\mathcal{T}$. ∎

Theorem III.13 $\mathcal{T} \notin \mathsf{RE}$*.*

Proof. Let $L \in \mathsf{RE} \setminus \mathsf{REC}$ and $F(y,z)$ be a formula that represents χ'_L. Let $x = e(F)$. It is easy to construct a WHILE program that given x and an $n \in \mathbb{N}$, computes an encoding e_n of the formula $F(n,1)$. Since

$$\chi'_L(n) = 1 \iff F(n,1) \text{ is true} \iff e_n \in \mathcal{T},$$

the mapping $n \mapsto e_n$ is a many-one reduction from L to $\mathcal{T}$. ∎

III.3 Proof systems

What is a proof? This is a subtle question whose answer is beyond the scope of this chapter. But here we only need two properties that without any doubt are properties that every proof system should have: The first is that the set

of all (encodings of) correct proofs should be decidable, that is, there is a WHILE program that can check whether a given proof is true. The second one is that there should be a total WHILE computable mapping that assigns to each proof the formula that is proven by this proof. Technically, proofs are finite words over some alphabet. Thus we can view them as natural numbers by using any "easy" injective mapping into $\mathbb{N}$.

Definition III.14 *A proof system for a set $L \subseteq \mathbb{N}$ is a tuple (P, F) such that*

1. *$P \subseteq \mathbb{N}$ is decidable and*
2. *$F : P \to L$ is a total WHILE computable function.*

We think of P of the set of (encodings of) proofs for the elements of L. The mapping F assigns each proof $p \in P$ the element of L that is proved by p.

Definition III.15 *A proof system (P, F) for L is complete if F is surjective.*

Theorem III.16 *There is no complete proof system for the set of all true arithmetic formulas $\mathcal{T}$.*

Proof. Assume there would be a complete proof system (P, F). The mapping

$$f : p \mapsto \begin{cases} F(p) & \text{if } p \in P \\ \text{undef.} & \text{otherwise} \end{cases}$$

is WHILE computable. By construction $\operatorname{im} f = \mathcal{T}$ and therefore $\mathcal{T} \in \mathsf{RE}$.. But this contradicts Theorem III.13. ∎

16 Turing machines

Turing machines are another model for computability. They were introduced by Alan Turing in the 1930s to give a mathematical definition of an algorithm. When Turing invented his machines, real computers were still to be built. Turing machines do not directly model any real computers or programming languages. They are abstract devices that model abstract computational procedures. The intention of Alan Turing was to give a formal definition of "intuitively computable"; rather than modeling computers, he modeled mathematicians. We will see soon that Turing machines and WHILE programs essentially compute the same functions. You can think of a Turing machine as an extension of a finite automaton: A finite automaton can only read symbols (and it reads every symbol only once, however, it turns out that this is not a real restriction). A Turing machine can also write symbols. The input is given on some "tape", which consists of cells, and each cell can store one symbol.

Why Turing machines?

Turing machines are *the* model for computations that you find in the textbooks. In my opinion, WHILE programs are easier to understand; it usually takes some time to get familiar with Turing machines.

I hope that at the end of this part you will see that it does not really matter whether one uses Turing machines or WHILE programs. All we need is a Gödel numbering, a universal Turing machine/WHILE program, and the ability to compute a Gödel number of the composition of two programs from the two individual Gödel numbers, i.e. an acceptable programming system. In theory, computer scientist are modest people.

16.1 Definition

A Turing machine M has a *finite control* and a number, say k, of *tapes*. The finite control is in one of the states from a set of states Q. Each of the tapes consists of an infinite number of cells and each cell can store one symbol from a finite alphabet Γ, the *tape alphabet*. (Here, finite alphabet is just a fancy word for finite set.) Γ contains one distinguished symbol, the *blank*

$\square$. Each tape is two-sided infinite.[1] That is, we can formally model it as a function $T : \mathbb{Z} \to \Gamma$ and $T(i)$ denotes the content of the ith cell. Each tape has a *head* that resides on one cell. The head can be moved back and forth on the tape, in a cell by cell manner. Only the content of the cells on which the heads currently reside can be read by M. In one step,

1. M reads the content of the cells on which its heads reside,
2. then M may change the content of these cells.
3. M moves each head either one cell to the left, not at all, or one cell to the right.
4. Finally, it changes its state.

The behaviour of M is described by a transition function

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k.$$

δ can be a partial function. $\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma'_1, \ldots, \gamma'_k, r_1, \ldots, r_k)$ means that if M is in state q and reads the symbols $\gamma_1, \ldots, \gamma_k$ on the tapes $1, \ldots, k$, then it will enter state q', replace the symbol γ_1 by γ'_1 on the first tape, γ_2 by γ'_2 on the second tape, etc., and move the heads as given by $r_1, \ldots, r_k$. (L stands for "left", S for "stay", and R for "right". If the head stands in position i of the tape, then "left" means that the head moves to position $i-1$ and "right" means that it moves to position $i+1$.) If $\delta(q, \gamma_1, \ldots, \gamma_k)$ is undefined, then M halts. Figure 16.1 is a schematic drawing of a Turing machine. I do not know whether it is really helpful, but every book on Turing machines contains such a drawing.

Definition 16.1 *A k-tape Turing machine M is described by a tuple $(Q, \Sigma, \Gamma, \delta, q_0)$ where:*

1. *Q is a finite set, the set of* states.
2. *Σ is a finite alphabet, the* input alphabet.
3. *Γ is a finite alphabet, the* tape alphabet. *There is a distiguished symbol $\square \in \Gamma$, the* blank. *We have $\Sigma \subseteq \Gamma \setminus \{\square\}$.*
4. *$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k$ is the* transition function.
5. *$q_0 \in Q$ is the* start state.

[1] In some textbooks, the tapes are only one-sided infinite. As we will see soon, this does not make any difference.

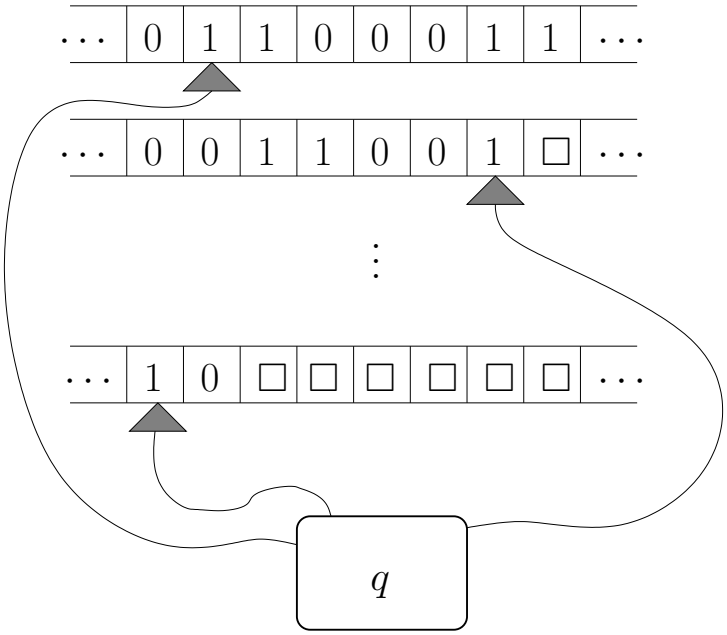


Figure 16.1: A Turing machine

In the beginning, all tapes are filled with blanks. The only exception is the first tape; here the input is stored. The input of a Turing machine is a string $w \in \Sigma^*$ where $\Sigma \subseteq \Gamma \setminus \{\square\}$ is the input alphabet. It is initially stored in the cells $0, \ldots, |w| - 1$ of the first tape. All heads stand on the cell with number 0 of the corresponding tape. The Turing machine starts in its start state q_0 and may now perform one step after another as described by the transition function δ.

Example 16.2 *Let us consider a first example: We will construct a 1-tape Turing machine INC that increases a given number in binary by* 1*. We assume that the head stands on the bit of lowest order and that in the end, the head will stop there again. The lowest order bit stands on the left and the highest order bit on the right, which is a bit unusual, but this will simplify further constructions. What does INC do? If the lowest order bit is a* 0*, then INC replaces it by a* 1 *and is done. If the lowest order bit is* 1*, then INC replaces it by a* 0*. This creates a carry and INC goes one step to the right and repeats this process until it finds a* 0 *or a* $\square$*. The latter case occurs if we add* 1 *to a number of the form* $111\ldots1$*.*

INC has three states add*,* back*, and* stop*. The state* add *is the start state. The input alphabet is* $\Sigma = \{0, 1\}$ *and the tape alphabet is* $\Gamma = \{0, 1, \square\}$*. The transition function is given by the following table:*

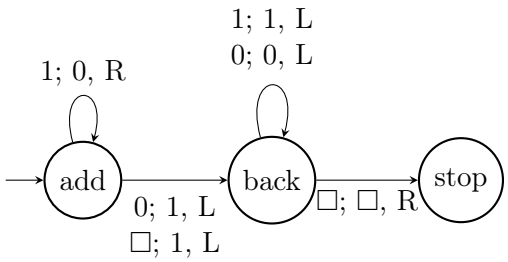


Figure 16.2: The transition diagram of the Turing machine INC from Example 16.2.

	0	1	□
add	(back, 1, L)	(add, 0, R)	(back, 1, L)
back	(back, 0, L)	(back, 1, L)	(stop,□, R)
stop	—	—	—

Above, "—" stands for undefined. In the state add*, INC goes to the right replacing every* 1 *by a* 0 *until it finds the first* 0 *or* □*. This* 0 *or* □ *is then replaced by a* 1 *and INC enters the state* back*. In the state* back*, INC goes to the left leaving the content of the cells unchanged until it finds the first* □*. It goes one step to the right and is done.*

Instead of a table, a transition diagram *is often more understandable. We have already encountered them when we studied finite automata. Figure 16.2 show this diagram for the Turing machine of this example. The states are drawn as circles and an arrow from q to q' with the label "$\alpha; \beta, r$" means that if the Turing machine is in state q and reads α, then it goes to state q', writes β, and moves its head as given by $r \in \{L, S, R\}$.*

16.2 Configurations and computations

A computation of a finite automaton is a sequence of states. The current state of a finite automaton completely determines its future behaviour. This is not completely true, since the position of a state in a computation also tells us, how many symbols of the input have been read by the automaton. So the current state and the rest of the input tells us the future behaviour of the automaton. We want to define something similar for Turing machines.

While computing, a Turing machine can change its state, change the positions of the heads, and the content of the tapes. The state together with the positions of the heads and the content of the tapes is called a *configuration*. While a tape of a Turing machine is potentially infinite, within any finite number of steps, the machine can visit only a finite number of cells.

Only these cells can contain symbols other than $\square$. The only exeption is the first tape on which the input stands in unvisited cells at the beginning. So instead of modeling a tape as a function $\mathbb{Z} \to \Gamma$ (which is the same as a two-sided infinite word over Γ), we just store the relevant parts of the tape, i.e., the cells that have already been visited and—in the case of the first tape—the cells were the input is written.

Formally, a configuration C of a k-tape Turing machine is an element $(q, (p_1, x_1), \dots (p_k, x_k)) \in Q \times (\mathbb{N} \times \Gamma^*)^k$ such that $1 \leq p_\kappa \leq |x_\kappa|$ for all $1 \leq \kappa \leq k$. $q \in Q$ is the current state of M. $x_1, \dots, x_k$ is the content of the cells visited so far of the tapes $1, \dots, k$. p_κ denotes the position of the head of the tape κ, $1 \leq \kappa \leq k$. We store the position relatively, i.e., p_κ denotes the position within x_κ but not necessarily the absolute position on the tape.

The *start configuration* of $M = (Q, \Sigma, \Gamma, \delta, q_0)$ with input w is the configuration $(q_0, (1, w), (1, \square), \dots, (1, \square))$. The input w stands on the first tape and the head is on the first symbol of it. On all other tapes, only one cell has been visited so far (the one the head is residing on), and this cell necessarily contains a $\square$. We usually denote the start configuration by $\mathrm{SC}_M(w)$.

Let $C = (q, (p_1, x_1), \dots (p_k, x_k))$ and $C' = (q', (p'_1, x'_1), \dots (p'_k, x'_k))$ be two configurations and let $x_\kappa = u_\kappa \alpha_\kappa v_\kappa$ with $|u_\kappa| = p_\kappa - 1$ and $\alpha_\kappa \in \Gamma$ for $1 \leq \kappa \leq k$. In other words, α_κ is the symbol of the cell the head is residing on. Then C' is called a *successor* of C if C' is reached from C by one step of M. Formally this means that if $\delta(q, \alpha_1, \dots, \alpha_k) = (q', \beta_1, \dots, \beta_k, r_1, \dots, r_k)$, then we have for all $1 \leq \kappa \leq k$,

$$x'_\kappa = u_\kappa \beta_\kappa v_\kappa$$

and

$$p'_\kappa = \begin{cases} p_\kappa - 1 & \text{if } r_\kappa = L, \\ p_\kappa & \text{if } r_\kappa = S, \\ p_\kappa + 1 & \text{if } r_\kappa = R, \end{cases}$$

unless $p_\kappa = 1$ and $r_\kappa = L$ or $p_\kappa = |x_\kappa|$ and $r_\kappa = R$. In the latter two cases, M is visiting a new cell. In these cases, we have to extend x_κ by one symbol. If $p_\kappa = 1$ and $r_\kappa = L$, then

$$x'_\kappa = \square \beta_\kappa v_\kappa$$

and

$$p'_\kappa = 1.$$

If $p_\kappa = |x_\kappa|$ and $r_\kappa = R$, then

$$x'_\kappa = u_\kappa \beta_\kappa \square$$

and

$$p'_\kappa = |x_\kappa| + 1.$$

We denote the fact that C' is a successor of C by $C \vdash_M C'$. Note that by construction, each configuration has at most one successor. We denote the reflexive and transitive closure of the relation $\vdash_M$ by $\vdash_M^*$, i.e., $C \vdash_M^* C'$ iff there are configurations $C_1, \ldots, C_\ell$ for some ℓ such that $C \vdash_M C_1 \vdash_M \ldots \vdash_M C_\ell \vdash_M C'$. If M is clear from the context, we will often omit the subscript M.

A configuration that has no successor is called a *halting configuration*. A Turing machine M halts on input w iff $\mathrm{SC}_M(w) \vdash_M^* C_t$ for some halting configuration C_t. (Note again that if it exists, then C_t is unique.) Otherwise M does not halt on w. If M halts on w and C_t is a halting configuration, we call a sequence $\mathrm{SC}_M(w) \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_t$ a *computation* of M on w. If M does not halt on w, then the corresponding computation is infinite.

Assume that $\mathrm{SC}_M(w) \vdash_M^* C_t$ and $C_t = (q, (p_1, x_1), \ldots, (p_k, x_k))$ is a halting configuration. Let $i \leq p_1$ be the largest index such that $x_1(i) = \Box$. If such an index does not exist, we set $i = 0$. In the same way, let $j \geq p_1$ be the smallest index such that $x_1(j) = \Box$. If such an index does not exist, then $j = |x_1| + 1$. Let $y = x_1(i+1)x_1(i+2)\ldots x_1(j-1)$. In other words, y is the word that the head of tape 1 is standing on. y is called the *output* of M on w. (This choice is fairly arbitrary, you can find other definitions in various text books, but they are all equivalent. For instance, you could think of an extra output tape, that is write-only and the Turing machine has to print its output there.)

16.3 Functions and languages

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0)$ computes a (partial) function $\varphi_M : \Sigma^* \to (\Gamma \setminus \{\Box\})^*$ defined by

$$\varphi_M(w) = \begin{cases} \text{the output of } M \text{ on } w & \text{if } M \text{ halts on } w, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Definition 16.3 *A function $f : \Sigma^* \to \Sigma^*$ is* Turing computable*, if $f = \varphi_M$ for some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0)$.*

We also want to define decidable languages. We could call a language $L \subseteq \Sigma^*$ decidable if its characteristic function $\Sigma^* \to \{0, 1\}$ is Turing computable. But this has the problem that 0 or 1 might not be elements of Σ. So we either have to put 0 and 1 into Σ or we have to identify two symbols of Σ with 0 and 1. While this works, there is a more elegant way: Like we did for finite automata, we will partition the states into accepting and rejecting states. A Turing machine is now described by a 6-tuple

$(Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$.[2] $Q_{\text{acc}} \subseteq Q$ is called the set of accepting states. A halting configuration $(q, (p_1, x_1), \ldots, (p_k, x_k))$ is called an *accepting configuration* if $q \in Q_{\text{acc}}$. Otherwise it is called a *rejecting configuration.*

Definition 16.4 *Let $L \subseteq \Sigma^*$ be a language.*

1. *A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$* recognizes *a language $L \subseteq \Sigma^*$, if for all $w \in L$, the computation of M on w ends in an accepting configuration and for all $w \notin L$, the computation does not end in an accepting configuration (i.e., it either ends in a rejecting configuration or M does not halt on w).*
2. *M* decides *L, if in addition, M halts on all $w \notin L$.*
3. *If M is a Turing machine then we denote by $L(M)$ the language recognized by M.*

Does the alphabet matter?

For WHILE programs, languages are always a subset of $\mathbb{N}$. For Turing machines, languages are subsets of Σ^* and the alphabet Σ may vary. From an esthetic point of view, this might be unpleasant. However, we can always restrict ourselves to the alphabet $\{0, 1\}$ by using a fixed length encoding of the symbols of Σ by binary strings.

[2] So if we want to compute a function, a 5-tuple is sufficient to describe the Turing machine. If we want to decide or recognize languages, then we take a 6-tuple. We could also always take a 6-tuple and simply ignore the accepting states if we compute functions.

17 Examples, tricks, and syntactic sugar

Understanding a Turing machines is like learning to code. In the beginning, even writing a function consisting of ten lines of code is hard. Once you got experienced, it is sufficient to specify the overall structure of your program and then filling the functions with code is an easy tasks. The same is true for Turing machines; though we will not fill in the technical details after specifying the overall structure since we do not want to sell Turing machines.

In the beginning, however, we will do low level descriptions of Turing machines, that is, we will write down the transition functions explicitly.

17.1 More Turing machines

Here are some example Turing machines that do some simple tasks. We will need them later on.

The Turing machine ERASE in Figure 17.1 erases the cells to the right of the head until the first blank is found.

The machine COPY in Figure 17.2 is a two-tape Turing machine. Itcopies the content of the first tape to the second tape provided that the second tape is empty. This is done in the state copy. Once the first blank on the first tape is reached, the copy process is finished and the Turing machine moves the heads back to the left-most symbol.

Figure 17.3 shows the Turing machine COMPARE. In the state zero?, it moves its head to the right until either a 1 or a □ is found. In the first case, the content is not zero. In the second case, the content is zero. In both states backn and backy, we go back to the left until we find the first blank. We use two different states for the same thing since we also have to store in

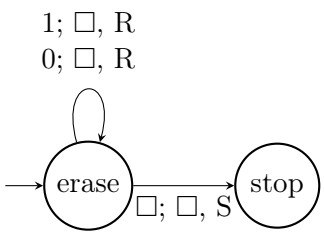


Figure 17.1: The Turing machine ERASE

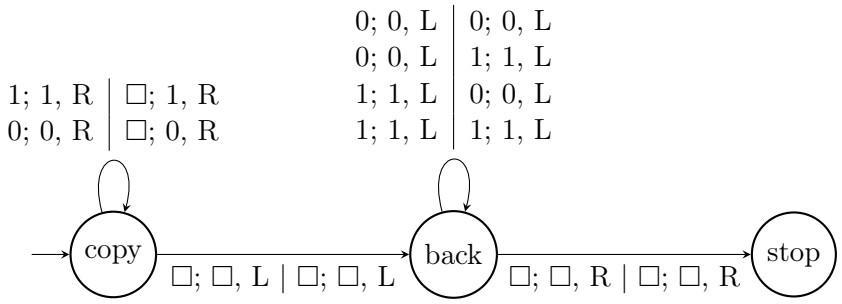


Figure 17.2: The Turing machine COPY

the state whether the content is zero or not. The Turing machine stops in the state yes or no.

Exercise 17.1 *Construct a Turing machine DEC that decreases the content of the tape by* 1 *if the content is* > 0.

17.2 Some techniques and tricks

17.2.1 Concatenation

Although Turing machines are very unstructured objects per se, it is quite useful to think about them in a structured way. For instance, assume we want to test whether the content of the tapes represents some string in binary that is either zero or one. We first run the Turing machine DEC of Exercise 17.1. After that the content of the tape has been decreased, it is zero if and only if it was zero or one before. Thus we can now run the Turing machine COMPARE. This new "concatenated" Turing machine can easily be constructed from DEC and COMPARE: First rename the states of COMPARE, then make a new Turing machine by throwing the states of both together. On the states of DEC, the Turing machine behaves like DEC, with one exception: Whenever DEC wants to enter stop, it enters the starting state of COMPARE instead. On the states of COMPARE, the Turing machine behaves like COMPARE. Figures 17.4 shows a schematic drawing that we will use in the following.

17.2.2 Loops

If you concatenate Turing machines "with themselves", you get loops. For instance, if you want to have a counter on some tape that is decreased by one until zero is reached, we can easily achieve this by concatenating the machines DEC and COMPARE as depicted in Figure 17.5.

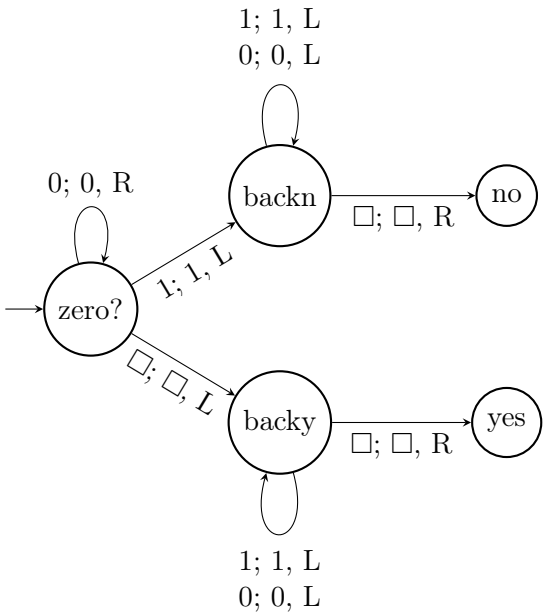


Figure 17.3: The Turing machine COMPARE

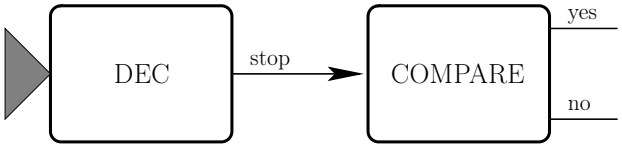


Figure 17.4: The concatenated Turing machine. The triangle to the left of DEC indicates that the starting state of DEC is the starting state of the new Turing machine. The arrow labeled with stop means that whenever DEC wants to enter the state stop of DEC, it enters the starting state of COMPARE instead. The lines labeled with yes and no mean that yes and no are the two halting states of COMPARE. Note that this concatenation works well in particular because the Turing machines move the head back to the first cell of the tape.

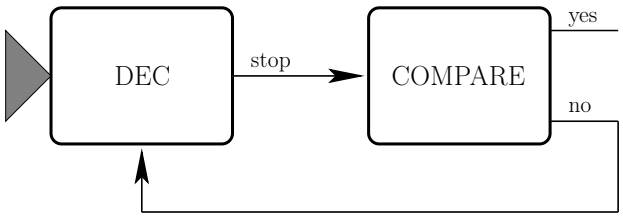


Figure 17.5: The counting Turing machine. DEC is executed; after this, the content of the tape is compared with zero. If the state no is entered, the machine enters again the starting state of DEC and decrease the content again. If the content of the tape is zero, then the Turing machine stops in the state yes.

17.2.3 Marking of cells

All the machines INC, COPY, COMPARE go back to the left end of the string on the tape. They can find this end because it is marked by a $\square$. What if a Turing machine wants to find a position somewhere in the middle of the string. In this case, it can "leave a marker" there. For this, we enlarge the tape alphabet and add for each $\gamma \in \Gamma$ a new symbol $\bar{\gamma}$. The Turing machine can replace the current symbol γ by the symbol $\bar{\gamma}$ and move its head somewhere else. It can find this position by going back and scanning for a symbol that is not in the original alphabet Γ. It then replaces this symbol $\bar{\gamma}$ by γ and continues its computation. (Leaving more than one marker in this way per tape could mean trouble!)

17.2.4 Storing information in the state

Have another look at the Turing machine COMPARE. After it reached the first 1 or the first $\square$ it has to go back to the beginning of the string. In both cases, COMPARE has to do the same thing: going back! But it has to remember whether it found a 1 or a $\square$. Therefore, we need two states for going back, backn and backy. One usually says that "the Turing machine goes back and stores in its state whether it found a 1 or a $\square$". If there is more to store, say an element from a set S, then it is more convenient to take the cartesian product $\{q\} \times S$ as states. In our example, we could e.g. also have used the elements from $\{\mathsf{back}\} \times \{1, \square\}$ as states.

17.2.5 Parallel execution

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ and $M' = (Q', \Sigma, \Gamma, \delta', q_0')$ be two Turing machines with k and k' tapes. We can construct a Turing machine N with $k+k'$ tapes which simulates M and M' in parallel as follows: N has states $Q \times Q'$ and

starting state (q_0, q_0'). The transition function of N,

$$\Delta : (Q \times Q') \times \Gamma^{k+k'} \to (Q \times Q') \times \Gamma^{k+k'} \times \{L, S, R\}^{k+k'},$$

is defined by

$$\Delta((q, q'), \gamma_1, \ldots, \gamma_{k+k'}) = ((p, p'), \alpha_1, \ldots, \alpha_k, \alpha_1', \ldots, \alpha_{k'}', r_1, \ldots, r_k, r_1', \ldots, r_{k'}')$$

if

$$\begin{aligned} \delta(q, \gamma_1, \ldots, \gamma_k) &= (p, \alpha_1, \ldots, \alpha_k, r_1, \ldots, r_k) \text{ and} \\ \delta'(q', \gamma_{k+1}', \ldots, \gamma_{k+k'}') &= (p', \alpha_1', \ldots, \alpha_{k'}', r_1', \ldots, r_{k'}') \end{aligned}$$

are both defined. If only one is defined, say $\delta(q, \gamma_1, \ldots, \gamma_k)$, then

$$\Delta((q, q'), \gamma_1, \ldots, \gamma_{k+k'}) = ((p, q'), \alpha_1, \ldots, \alpha_k, \gamma_{k+1}, \ldots, \gamma_{k+k'}, r_1, \ldots, r_k, S, \ldots, S)$$

The other case is defined symmetrically. If both are undefined, then $\Delta((q, q'), \gamma_1, \ldots, \gamma_{k+k'})$ is undefined, too. On the first k tapes, N behaves like M, on the other k' tapes, N behaves like M'. If one machine stops, then N does not move its head on the corresponding tapes anymore and just writes the symbols that it reads all the time. If the second machine stops, too, then N stops. (Of course, since N gets its input on the first tape, M' is simulated on the empty tape. If we want to simulate M and M' on the same input, then we have to copy the input from the first tape to the $(k+1)$th tape, using for instance COPY, before N starts with the simulation).

Here is one application: take M to be any machine and M' is the machine from Figure 17.5. Modify the function Δ such that M' is simulated normally, but M only executes a step when M' changes it state from the state no to its start state. In this way, M executes as many steps as given by the counter in the beginning. We will need this construction later on.

17.3 Syntactic sugar

For more complex Turing machines, describing them by transition diagrams is a boring task. So after some training, we will go on by describing Turing machines by sentences in natural language. Whenever you formulate such a sentence, you should carefully think how a Turing machine actually would do the thing that you are describing. Here is a description of the machine INC:

Input: $x \in \{0,1\}^*$, viewed as a number in binary.
Output: x increased by 1

1. Go to the right and replace every 1 by a 0 until you reach the first 0 or $\square$.
2. Write a 1 and go back to the right to the beginning of the string.

Once we got even more experienced with Turing machines, we could even write:

Input: $x \in \{0,1\}^*$, viewed as a number in binary.
Output: x increased by 1

1. Increase the content of the tape by 1.

Warning!!! Although the example above suggests it, the sentence "The Turing machine produces the desired ouput" is in general not an adequate description of a Turing machine. Above, it is, because the Turing machine INC does such a simple job. More complex jobs, require more detailed descriptions.

If in doubt ...

... whether your description of a Turing machine is o.k. always ask yourself the following question: Given the description, can I write a C program that gets k `char*` as input and does the same.

(JAVA is also fine.)

17.4 Further exercises

Exercise 17.2 *Instead of a two-sided infinite tape, you can also find Turing machines with a one-sided infinite tape in the literature. Such a tape can be modelled by a function $T : \mathbb{N} \to \Gamma$. There is a distinguished symbol & that marks the end of each tape. Initially, every tape is filled with blanks except the 0th cell, which is filled with a &. The first tape contains the input x in the cells $1, 2, \ldots, |x|$. Every Turing machine with one-sided infinite tapes has to obey the following rules: If it does not read a &, it cannot write & on this tape. If it reads a &, it has to write a & on this tape and must not move its*

head to the left on this tape. In this way, it can never leave the tape to the left.

Show that every Turing machine with two-sided infinite tapes can be simulated by a Turing machine with one-sided infinite tapes. Try not to increase the number of tapes!

Exercise 17.3 *Consider the following language*

$$\text{COPY} = \{w\#w \mid w \in \{0,1\}^*\}$$

over the alphabet $\{0, 1, \#\}$.

1. *Modify the COPY machine such that it recognizes the language* COPY. *Although we did not define running time for Turing machines so far, how many steps does the Turing machine do on inputs of length* $m = 2n + 1$*?*

2. *Can you recognize* COPY *with a one-tape Turing machine? How many steps does the Turing machine do on inputs of length* m*?*

18 Church–Turing thesis

18.1 WHILE versus Turing computability

In this chapter, we want to show that

WHILE computable equals Turing computable.

But there is of course a problem. WHILE programs compute functions $\mathbb{N}^s \to \mathbb{N}$ whereas Turing machines compute functions $\Sigma^* \to \Sigma^*$. To make things a little easier, we can restrict ourselves to functions $\mathbb{N} \to \mathbb{N}$, since we can use a pairing function. For Turing machines, we use the input alphabet $\Sigma = \{0, 1\}$.

18.1.1 $\mathbb{N}$ versus $\{0, 1\}^*$

We have to identify natural numbers with words over $\{0, 1\}$ and vice versa. For $y \in \mathbb{N}$, let $\text{bin}(y) \in \{0, 1\}^*$ denote the binary expansion of y without any leading zeros. (In particular, $0 \in \mathbb{N}$ is represented by the empty word ε and not by 0.) The function $\text{bin} : \mathbb{N} \to \{0, 1\}^*$ is an injective mapping—two different numbers have different binary expansions—but it is not surjective, since we do not cover strings with leading zeros. Not bad and this would work, but for esthetic reasons, we want to have a bijection between $\mathbb{N}$ and $\{0, 1\}^*$. Consider the following mapping $\{0, 1\}^* \to \mathbb{N}$: Append a 1 to a given $x \in \{0, 1\}^*$. This is an injective mapping from $\{0, 1\}^*$ to the subset of all binary expansions without leading zeros of some natural number. Since we do not have leading zeros, the function that maps such a binary expansion to the corresponding natural number is also injective. The combination of both gives an injective mapping $\{0, 1\}^* \to \mathbb{N}$. It is also surjective? No, the smallest number that we get in the image is 1, by appending 1 to the empty word ε. So here is the next attempt:

1. Append a 1 to the word $x \in \{0, 1\}^*$.

2. View this string $1x$ as some binary expansion. Let n be the corresponding number, i.e, $\text{bin}(n) = 1x$.

3. Subtract 1 from n.

We call the mapping $\{0,1\}^* \to \mathbb{N}$ that we get this way cod. More compactly, we can write $\mathrm{cod}(x) = \mathrm{bin}^{-1}(1x) - 1$. (Note that we can write bin^{-1}, since $1x$ is in the image of bin.)

Exercise 18.1 *Show that* cod *is indeed bijective.*

18.1.2 $\mathbb{N} \to \mathbb{N}$ versus $\{0,1\}^* \to \{0,1\}^*$

Now, if $f : \mathbb{N} \to \mathbb{N}$ is a function, then $\hat{f} : \{0,1\}^* \to \{0,1\}^*$ is defined as follows:

$$\hat{f}(x) = \mathrm{cod}^{-1}(f(\mathrm{cod}(x))) \quad \text{for all } x \in \{0,1\}^*.$$

Or, in other words, the following diagram commutes:

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{f} & \mathbb{N} \\ \downarrow \mathrm{cod}^{-1} & & \downarrow \mathrm{cod}^{-1} \\ \{0,1\}^* & \xrightarrow{\hat{f}} & \{0,1\}^* \end{array}$$

Conversely, if $g : \{0,1\}^* \to \{0,1\}^*$, then $\hat{g} : \mathbb{N} \to \mathbb{N}$ is defined by

$$\hat{g}(n) = \mathrm{cod}(g(\mathrm{cod}^{-1}(n)) \quad \text{for all } n \in \mathbb{N}.$$

In other words,

$$\begin{array}{ccc} \{0,1\}^* & \xrightarrow{g} & \{0,1\}^* \\ \downarrow \mathrm{cod} & & \downarrow \mathrm{cod} \\ \mathbb{N} & \xrightarrow{\hat{g}} & \mathbb{N} \end{array}$$

Exercise 18.2 *Show the following: For every $f : \mathbb{N} \to \mathbb{N}$ and $g : \{0,1\}^* \to \{0,1\}^*$, $\hat{\hat{f}} = f$ and $\hat{\hat{g}} = g$. ("$\hat{\cdot}$ is self-inverse.")*

Remark 18.1 *The mapping* cod *is not too natural. For instance, with $f : \mathbb{N} \to \mathbb{N}$, we could associate the mapping $\mathrm{bin}(x) \mapsto \mathrm{bin}(f(x))$. But if* cod *is a bijection, then we have the nice property that $\hat{\cdot}$ is self-inverse.*

Exercise 18.3 *Show that* cod *and* cod^{-1} *are functions that are easy to compute. In particular:*

1. *Write a WHILE program that, given an $n \in \mathbb{N}$, computes the symbols of $\mathrm{cod}^{-1}(n)$ and stores them in an array.*
2. *Construct*[1] *a Turing machine that given $x \in \{0,1\}^*$, writes $\mathrm{cod}(x)$ many 1's on the first tape.*

So the only reason why a WHILE program or Turing machine cannot compute cod *or* cod^{-1} *is that they cannot directly store elements from $\{0,1\}^*$ or $\mathbb{N}$, respectively.*

[1] This is quite funny. While both WHILE programs and Turing machines are mathematical objects, we *write* WHILE programs but *construct* Turing machines.

18.1.3 Pairing functions

We also need a pairing function for $\{0,1\}^*$, i.e., an injective mapping $[.,.] : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. If we take our pairing function $\langle .,. \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, then of course,

$$(x,y) \mapsto \mathrm{cod}^{-1}(\langle \mathrm{cod}(x), \mathrm{cod}(y) \rangle)$$

is a bijection $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$.

But there are nicer ways of defining a pairing function for $\{0,1\}^*$. The easiest way would be to concatenate the strings x and y. But then we do not know where x ends and y starts. We could use some special symbol $\#$ and write it between x and y, but then we would have to enlarge the input alphabet (which is not a tragedy, but we do not want to do it here). There is another way: Let $\beta(x) = x_1 0 x_2 0 \ldots x_{|x|} 0$, i.e, we insert a 0 after every symbol of x. From $\beta(x)11y$, we can get x and y back, since the first pair 11 marks the end of $\beta(x)$. The length of this encoding is $2|x| + |y| + 2$. We can get a shorter one by mapping (x,y) to $\beta(\mathrm{bin}(|x|))11xy$. By scanning for the first pair 11, we can divide the string into $\beta(\mathrm{bin}(|x|))$ and xy. From the first, we can reconstruct $|x|$. Once we know this, we can get x and y from x. The length of this encoding is $\leq |x| + |y| + 2\log|x| + 2$.

Exercise 18.4 *Try to get even shorter encodings in this way. What is the shortest that you can get?*

18.2 GOTO programs

It turns out that it is useful to introduce some intermediate concept, *GOTO programs*. GOTO programs have variables and the same simple statements as WHILE programs but instead of a while loop, there is a goto statement. Furthermore, all the lines are numbered.

Formally, a GOTO program is a sequence $(1, s_1), (2, s_2), \ldots, (m, s_m)$ where each s_μ is a statement of the form

1. $x_i = x_j + x_k$ or
2. $x_i = x_j - x_k$ or
3. $x_i := c$ or
4. **if** $x_i \neq 0$ **then goto** λ

The semantics of the first three statements is the same as for WHILE programs. After the μth statement is executed, the program goes on with the $(\mu+1)$th statement. The only execption is the *conditional jump* **if** $x_i \neq 0$ **then goto** λ. If the content of x_i is zero, then we go on with the $(\mu+1)$th statement, otherwise, we go on with statement λ. If we ever reach a line that

does not exist—either by jumping to a nonexisting line or by executing the last statement s_m and going to line $m+1$—the program stops. The content of x_0 is the value computed by the program. As for WHILE programs, the input is stored in the first s variables. The function $\mathbb{N}^s \to \mathbb{N}$ computed by a GOTO program P is denoted by—surprise—φ_P.

Exercise 18.5 *Show how to simulate an* unconditional jump *in GOTO. (An unconditional jump, we denote it by* **goto** λ*, always jumps to line* λ *no matter what.)*

Exercise 18.6 *Give a precise mathematical formulation of the semantics of GOTO programs. A state should consist of a natural number, which stores the current line to be executed, and a tuple/sequence with finite support of natural numbers, which stores the content of the variables. Construct a function* Φ_P *that maps a state* (i, V) *to the state that is reached after executing line* i*.*

Every while loop can be simulated by a goto statement. If should be fairly obvious that

while $x_i \neq 0$ **do**
  P
od

is simulated by

1: **if** $x_i \neq 0$ **then goto** 3
2: **goto** 5
3: P
4: **goto** 1
5: ...

The use of the labels is a little bit sloppy. The program P in general has more than one line, so the label of the statement **goto** 1 is usually larger. Furthermore, we do not write tuples but lines and separate labels and statements by ":". We get the following theorem.

Lemma 18.2 *For every WHILE program* P *there is a GOTO program* Q *with* $\varphi_P = \varphi_Q$*.*

18.3 Turing machines can simulate GOTO programs

Lemma 18.3 *Let* $f : \mathbb{N} \to \mathbb{N}$*. If* f *is GOTO computable, then* $\hat{f}$ *is Turing computable.*

Proof. Assume that f is GOTO computable. Let $P = (1, s_1), \ldots, (m, s_m)$ be a GOTO program computing f. It is fairly easy to see we can restrict the simple statements of GOTO programs to $x_i{+}{+}$, $x_i{-}{-}$, and $x_i := 0$.

Assume that P uses variables $x_0, \ldots, x_\ell$. Our Turing machine uses $\ell + 1$ tapes. Each tape stores the content of one of the registers in binary. The input for the Turing machine is $\mathrm{cod}^{-1}(n)$. For the simulation, it is easier to have $\mathrm{bin}(n)$ on the tape. We get it by appending a 1 to $\mathrm{cod}^{-1}(n)$ and then subtracting 1 by using the Turing machine DEC.

We will use the Turing machines INC, DEC, ERASE, and COMPARE (see Section 17) to construct a Turing machine M that simulates P, more precisely, computes $\hat{f}$.

For each instruction (μ, s_μ), we have a state q_μ. The invariant of the simulation will be that whenever the Turing machine is in one of these states, the content of the tapes correspond to the content of the registers before executing the instruction s_μ and all heads stand on the left-most symbol that is not a blank. (The lowest order bit is standing on the left.)

Figure 18.1 shows an example of the construction for the program

1: **if** $x_0 \neq 0$ **then goto** 3
2: $x_0{+}{+}$
3: ...

The arrow from the state q_0 to the box with the label COMPARE means that in q_0, M does nothing (i.e., writes the symbol that it reads and does not move its head) and enters the starting state of the machine COMPARE. The two arrows leaving this box with the labels yes and no mean that from the states yes and no of COMPARE, we go to the states q_3 and q_2,respectively. The Turing machine COMPARE is only a 1-tape Turing machine. It can be easily extended it to an $(\ell + 1)$-tape machine that only works on the tape corresponding to x_0. The same has to be done for the machine INC and so on. From the example it should be clear how the general construction works. For each instruction s_μ, M goes from q_μ to a copy of one of the Turing machines that simulates the instruction $x_i{+}{+}$, $x_i{-}{-}$, $x_i := 0$ or **if** $x_i \neq 0$ **then goto** λ. From the halting state(s) of these machines, M then goes to $q_{\mu+1}$, the only exception being the conditional jump.

It should be clear from the construction that the simulation is correct. To formally prove the correctness, it is sufficient to show the following statement:

Claim. Assume that P is in state (μ, V) and that $\Phi_P(\mu, V) = (\mu', V')$. If M is in state q_μ, the content of the tapes are $\mathrm{bin}(V(\lambda))$, $0 \leq \lambda \leq \ell$, and the heads are standing on the lowest order bits of $\mathrm{bin}(V(\lambda))$, then the next state from $q_1, \ldots, q_m$ that M will enter will be $q_{\mu'}$. At this point, the content of the tapes are $\mathrm{bin}(V'(\lambda))$, $1 \leq \lambda \leq \ell$, and the heads are standing on the lowest order bits of $\mathrm{bin}(V(\lambda))$.

From this claim, the correctness of the simulation follows immediately. When M stops, we have to transform the binary expansion $b = \mathrm{bin}(n)$ on tape 1 back into $\mathrm{cod}^{-1}(n)$, which is easy. ■

Exercise 18.7 *Give a detailed description of the general construction.*

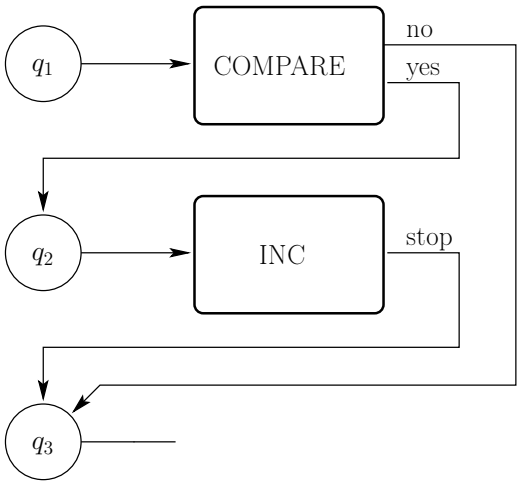


Figure 18.1: The simulating Turing machine for the example GOTO program.

Exercise 18.8 *Prove the claim in the proof of Lemma 18.3*

18.4 WHILE programs can simulate Turing machines

Lemma 18.4 *Let $g : \{0,1\}^* \to \{0,1\}^*$. If g is Turing computable, then $\hat{g}$ is WHILE computable.*

Proof. Let $M = (Q, \{0,1\}, \Gamma, \delta, q_0)$ be a k-tape Turing machine that computes g. By renaming states, we can assume that $Q = \{1, 2, \ldots, q\}$. We represent the symbols of Γ by the numbers $0, 1, \ldots, s-1$. By Exercise 17.2, we can assume that each tape is onesided infinite.

The content of each tape κ is stored in an array A_κ. $A_\kappa[i] = j$ means that the cell i contains the symbol that corresponds to j. Of course, A_κ always stores only a finite amount of data. Recall that the arrays that we created in WHILE can be made dynamic (Exercise **??**), so we can extend them whenever M visits a new cell. The variable p_κ contains the (absolute) position of the head.

In the beginning, we have to write $\mathrm{cod}^{-1}(x_0)$ into A_1. We can do this using the Turing machine constructed in Exercise 18.3.

The domain of δ is finite, thus we can hardwire the table of δ into our WHILE program. Then it is easy to simulate one step of M, we just have to update the corresponding cells of the array and adjust $p_1, \ldots, p_k$ and change the state. The simulation of one such step is embedded into a while loop of the form

while $\delta(q, a_1, \ldots, a_k)$ is defined **do** ... **od**

From the construction, it is clear that the simulation is correct. ■

From the proof above, we also get an new proof of the Kleene normal form for WHILE programs:

Alternative proof of Corollary 11.3. We convert a WHILE program into an equivalent GOTO program, then into an equivalent Turing machine, and finally back into an equivalent WHILE program. This WHILE program has only one while loop. All the other things, like simulating an array etc. can be done by FOR programs. ■

18.5 Church–Turing thesis

From Lemmas 18.2, 18.3, and 18.4, we get the following result.

Theorem 18.5 *Let $f : \mathbb{N} \to \mathbb{N}$. Then the following three statements are equivalent:*

1. *f is WHILE computable.*
2. *f is GOTO computable.*
3. *$\hat{f}$ is Turing computable.*

The Church–Turing thesis states that any notion of "intuitively computable" is equivalent to Turing computable (or WHILE computable, ...). The theorem above is one justification of the Church–Turing thesis. The Church–Turing thesis is not a statement that you could prove. It is a statement about the physical world we are living in. You can either accept the Church–Turing thesis or reject it. So far, the Church–Turing thesis seems to hold and it is widely accepted among computer scientists. Even a quantum computer would not change this. Quantum computers cannot compute more than Turing machines. Maybe they can do it faster, but this is another story. But who knows, maybe some day a brilliant (crazy?) physicist will come up with a device that decides the halting problem.

IV Primitive and μ-recursion

Historically, *primitive recursive* functions and *μ-recursive* functions are one of the first concepts to capture "computability".

IV.1 Primitive recursion

We are considering functions $\mathbb{N}^s \to \mathbb{N}$ for any $s \geq 1$.

Definition IV.1 *The set of all primitive recursive functions is defined inductively as follows:*

1. *Every* constant function *is primitive recursive.*
2. *Every* projection $p_i^s : \mathbb{N}^s \to \mathbb{N}$ *(mapping $(a_1, \ldots, a_s)$ to a_i) is primitive recursive.*
3. *The* successor function $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$ *defined by* $\mathrm{suc}(n) = n + 1$ *is primitive recursive.*
4. *If $f : \mathbb{N}^s \to \mathbb{N}$ and $g_i : \mathbb{N}^t \to \mathbb{N}$, $1 \leq i \leq s$, are primitive recursive, then their composition defined by*

$$(a_1, \ldots, a_t) \mapsto f(g_1(a_1, \ldots, a_t), \ldots, g_s(a_1, \ldots, a_t))$$

   is primitive recursive.

5. *If $g : \mathbb{N}^s \to \mathbb{N}$ and $h : \mathbb{N}^{s+2} \to \mathbb{N}$ are primitive recursive, then the function $f : \mathbb{N}^{s+1} \to \mathbb{N}$ defined by*

$$\begin{aligned} f(0, a_1, \ldots, a_s) &= g(a_1, \ldots, a_s) \\ f(n+1, a_1, \ldots, a_s) &= h(f(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s) \end{aligned}$$

   is primitive recursive. This scheme is called primitive recursion.

We want to show that primitive recursive functions are the same as FOR computable functions. Therefore, we first look at some fundamental functions that appear in FOR programs:

Example IV.2 *The function* $\mathrm{add}(x, y) = x + y$ *is primitive recursive. We have*

$$\begin{aligned} \mathrm{add}(0, y) &= y, \\ \mathrm{add}(x+1, y) &= \mathrm{suc}(\mathrm{add}(x, y)). \end{aligned}$$

Above, we did not write down the projections explicitly. The correct definition looks like this:

$$\begin{aligned}\mathrm{add}(0,y) &= p_1^1(y),\\ \mathrm{add}(x+1,y) &= \mathrm{suc}(p_1^3(\mathrm{add}(x,y),x,y)).\end{aligned}$$

Since this looks somewhat confusing, we will omit the projections in the following.

Exercise IV.1 *Show that the function* $\mathrm{mult}(x,y) = xy$ *is primitive recursive.*

Example IV.3 *The predecessor function defined by*

$$\mathrm{pred}(n) = \begin{cases} n-1 & \text{if } n > 0\\ 0 & \text{otherwise}\end{cases}$$

is primitive recursive. We have

$$\begin{aligned}\mathrm{pred}(0) &= 0,\\ \mathrm{pred}(n+1) &= n.\end{aligned}$$

This is a primitive recursion scheme.

Exercise IV.2 *Prove that the modified difference* $\mathrm{sub}(x,y) = \max\{x-y,0\}$ *is primitive recursive.*

IV.1.1 Bounded maximization

Finally, we will show that the pairing function $\langle .,. \rangle$ and "its inverses" π_1 and π_2 are primitive recursive. Recall that $\langle x,y\rangle = \frac{1}{2}(x+y)(x+y+1)+y$. From

$$\frac{1}{2}n(n+1) = \frac{1}{2}(n-1)n + n$$

we get a primitive recursion scheme for $\frac{1}{2}n(n+1)$ and from this function, we can easily get $\langle x,y\rangle$.

For the inverse functions, we need *bounded maximization*: Let P be a predicate on $\mathbb{N}$ and view P as a function $P : \mathbb{N} \to \{0,1\}$. Assume that P (as a function) is primitive recursive. We claim that

$$\text{bounded-max-}P(n) := \max\{x \le n \mid P(x) = 1\}$$

is primitive recursive. (If no such x exists, i.e., the maximum is undefined, we set bounded-max-$P(n) = 0$.) This can be seen as follows:

$$\begin{aligned}\text{bounded-max-}P(0) &= 0\\ \text{bounded-max-}P(n+1) &= \begin{cases} n+1 & \text{if } P(n+1) = 1\\ \text{bounded-max-}P(n) & \text{otherwise}\end{cases}\\ &= (1-P(n+1)) \cdot \text{bounded-max-}P(n) + P(n+1)\cdot(n+1).\end{aligned}$$

In the same way, we can see that the *bounded existential quantifier* defined by

$$\text{bounded-}\exists\text{-}P(n) := \begin{cases} 1 & \text{if there is an } x \leq n \text{ with } P(x) = 1 \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive:

$$\begin{aligned} \text{bounded-}\exists\text{-}P(0) &= P(0) \\ \text{bounded-}\exists\text{-}P(n+1) &= P(n+1) + \text{bounded-}\exists\text{-}P(n)(1 - P(n+1)) \end{aligned}$$

Above, P has only one argument. It is easy to see that for a predicate with s arguments,

$$\text{bounded-max}_i\text{-}P(x_1, \ldots, x_s) := \max\{x \leq x_i \mid P(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_s) = 1\}$$

is primitive recursive. In the same way, we can define bounded-$\exists_i$-P and show that it is primitive recursive.

With these quantifiers, we can easily invert the pairing function. Let $Q(x, y, z)$ be the predicate "$\langle x, y\rangle = z$". It is not too hard to see that this predicate is primitive recursive. Now we define:

$$\begin{aligned} \pi_1'(x', y', z') &:= \max\{x \leq x' \mid \exists y \leq y' : \ \langle x, y\rangle = z'\} \\ &= \max\{x \leq x' \mid \text{bounded-}\exists_2\text{-}Q(x, y', z')\} \\ &= \text{bounded-max}_1\text{-bounded-}\exists_2\text{-}Q(x', y', z'). \end{aligned}$$

Above, we use the fact, that $\langle ., . \rangle$ is a bijection, that is, given x and z, there is exactly one y with $\langle x, y\rangle = z$. The same holds when y and z is given. Now we can write:

$$\pi_1(z) = \pi_1'(z, z, z),$$

where we use the fact that $x, y \leq \langle x, y\rangle$ for all x, y. In the same way, we can prove that the projection π_2 is injective.

It immediately follows that also forming larger "pairs" $\langle a_1, \ldots, a_s\rangle$ and the corresponding inverse functions $\pi_1, \ldots, \pi_s$ are primitive recursive.

IV.1.2 Simultaneous primitive recursion

Let $g_i : \mathbb{N}^s \to \mathbb{N}$ and $h_i : \mathbb{N}^{s+t+1} \to \mathbb{N}$, $1 \leq i \leq t$, be primitive recursive. The functions $f_i : \mathbb{N}^{s+1} \to \mathbb{N}$, $1 \leq i \leq t$, defined by the *simultaneous primitive recursion scheme*

$$\begin{aligned} f_i(0, a_1, \ldots, a_s) &= g_i(a_1, \ldots, a_s), \\ f_i(n+1, a_1, \ldots, a_s) &= h_i(f_1(n, a_1, \ldots, a_s), \ldots, f_t(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s), \end{aligned}$$

for $i = 1, \ldots, t$, are primitive recursive. To see this, define f by

$$\begin{aligned} f(0,a) &= \langle g_1(\pi_1(a),\ldots,\pi_s(a)),\ldots,g_t(\pi_1(a),\ldots,\pi_s(a))\rangle \\ f(n+1,a) &= \langle h_1(\pi_1(f(n,\pi_1(a),\ldots,\pi_s(a))),\ldots, \\ &\qquad \pi_t(f(n,\pi_1(a),\ldots,\pi_s(a))),n,\pi_1(a),\ldots,\pi_s(a)),\ldots \\ &\quad h_t(\pi_1(f(n,\pi_1(a),\ldots,\pi_s(a))),\ldots, \\ &\qquad \pi_t(f(n,\pi_1(a),\ldots,\pi_s(a))),n,\pi_1(a),\ldots,\pi_s(a))\rangle. \end{aligned}$$

f is primitive recursive. By an easy induction on n, we can show that

$$\pi_i(f(n,a)) = f_i(n,\pi_1(a),\ldots,\pi_s(a)).$$

We can rewrite this as

$$f_i(n,a_1,\ldots,a_s) = \pi_i(f(n,\langle a_1,\ldots,a_s\rangle)).$$

Thus each f_i is primitive recursive.

IV.1.3 Primitive recursion versus for loops

The next lemma shows that for all FOR programs, there are primitive recursive functions that compute the values of the variables after executing P.

Lemma IV.4 *Let P be a FOR program with s inputs. Let ℓ be the largest index of a variable in P. Then there are primitive recursive functions $v_0,\ldots,v_\ell : \mathbb{N}^{\ell+1} \to \mathbb{N}$ such that*

$$(v_0(a_0,\ldots,a_\ell),\ldots,v_\ell(a_0,\ldots,a_\ell)) = \Phi_P(a_0,\ldots,a_\ell)$$

for all $a_0,\ldots,a_\ell \in \mathbb{N}$.

Proof. The proof is by structural induction.
Induction base: If P is $x_i := x_j + x_k$ then each v_λ is the projection on the λth component, except for v_i, which is $x_j + x_k$. Since modified subtraction and constant functions are primitive recursive, too, we can cover the cases $x_i := x_j - x_k$ and $x_i := c$ in the same way.
Induction step: If $P = [P_1; P_2]$, then by the induction hypothesis, there are primitive recursive functions $v_{i,0},\ldots,v_{i,\ell}$, $i = 1, 2$, such that

$$(v_{i,0}(a_0,\ldots,a_\ell),\ldots,v_{i,\ell}(a_0,\ldots,a_\ell)) = \Phi_{P_i}(a_0,\ldots,a_\ell), \qquad i = 1,2.$$

Since $\Phi_P = \Phi_{P_2} \circ \Phi_{P_1}$, we get that

$$v_\lambda(a_0,\ldots,a_\ell) = v_{2,\lambda}(v_{1,0}(a_0,\ldots,a_\ell),\ldots,v_{1,\ell}(a_0,\ldots,a_\ell))$$

for all $0 \le \lambda \le \ell$ and $a_0,\ldots,a_\ell \in \mathbb{N}$. Thus $v_0,\ldots,v_\ell$ are primitive recursive.

If $P =$ **for** x_i **do** P_1 **od**, then by the induction hypothesis, there are functions $v_{1,0}, \ldots, v_{1,\ell}$ such that

$$(v_{1,0}(a_0, \ldots, a_\ell), \ldots, v_{1,\ell}(a_0, \ldots, a_\ell)) = \Phi_{P_1}(a_0, \ldots, a_\ell). \tag{IV.1}$$

Define $u_0, \ldots, u_\ell$ by

$$\begin{aligned} u_\lambda(0, a_0, \ldots, a_\ell) &= a_\lambda \\ u_\lambda(n+1, a_0, \ldots, a_\ell) &= v_{1,\lambda}(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_0, \ldots, a_\ell)) \end{aligned}$$

for $0 \le \lambda \le \ell$. This is a simultaneous primitive recursion scheme. We claim that

$$(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_0, \ldots, a_\ell)) = \Phi_{P_1}^{(n)}(a_0, \ldots, a_\ell) \tag{IV.2}$$

for all $n, a_0, \ldots, a_\ell \in \mathbb{N}$. This claim is shown by induction on n.
Induction base: The case $n = 0$ is clear, since both sides are $(a_0, \ldots, a_\ell)$ in this case.
Induction step: We have

$$\begin{aligned} u_\lambda(n+1, a_0, \ldots, a_\ell) &= v_{1,\lambda}(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_1, \ldots, a_\ell)) \\ &= v_{1,\lambda}(\Phi_{P_1}^{(n)}(a_0, \ldots, a_\ell)) \\ &= \lambda\text{th entry of } \Phi_{P_1}^{(n+1)}(a_0, \ldots, a_\ell). \end{aligned}$$

The last equality follows from the induction hypothesis (IV.1).

Altogether, this shows (IV.2). We get v_λ by $v_\lambda(a_0, \ldots, a_n) = u_\lambda(a_i, a_0, \ldots, a_n)$.

■

Lemma IV.5 *For every primitive recursive function f, there is a FOR program P with $\varphi_P = f$.*

Proof. This proof is again by structural induction.
Induction base: Constant functions, projections, and the successor function are all FOR computable.
Induction step: If f is the composition of h and $g_1, \ldots, g_s$, then by the induction hypothesis, there are FOR programs P and $Q_1, \ldots, Q_s$ that compute h and $g_1, \ldots, g_s$. From this, we easily get a program that computes f.

If f is defined by

$$\begin{aligned} f(0, a_1, \ldots, a_s) &= g(a_1, \ldots, a_s), \\ f(n+1, a_1, \ldots, a_s) &= h(f(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s), \end{aligned}$$

then there are programs P and Q that compute h and g, respectively. Now the following program computes $f(a_0, a_1, \ldots, a_s)$:

1: $x_0 := g(a_1, \ldots, a_s)$;
2: **for** a_0 **do**
3: $\quad x_0 := h(x_0, a_0, a_1, \ldots, a_s)$
4: **od**

We saw how to simulate function calls of FOR computable functions. ■

Theorem IV.6 *A function f is primitive recursive iff it is FOR computable.*

Proof. The "$\Leftarrow$"-direction is Lemma IV.5. The function v_0 of Lemma IV.4 is the function that is computed by the program P. This shows the other direction. ■

IV.2 μ-recursion

The μ-operator allows unbounded search.

Definition IV.7 *Let $f : \mathbb{N}^{s+1} \to \mathbb{N}$. The function $\mu f : \mathbb{N}^s \to \mathbb{N}$ is defined by*

$$\mu f(a_1, \ldots, a_s) = \min\{n \mid f(n, a_1, \ldots, a_s) = 0 \text{ and } \text{for all } m < n,\ f(m, a_1, \ldots, a_s) \text{ is defined}\}.$$

Definition IV.8 *The set of all* μ-recursive functions *is defined inductively as in Defition IV.1 except that the set is closed under μ-recursion instead of primitive recursion.*

Theorem IV.9 *A function is μ-recursive iff it is WHILE computable.*

Proof. This proof is just an "add-on" to the proof of Theorem IV.6.

For the "$\Rightarrow$"-direction, we just have to consider one more case in the proof of Lemma IV.5. If $f = \mu g$ for some μ-recursive function $g : \mathbb{N}^{s+1} \to \mathbb{N}$, then we have to show that f is WHILE computable provided that g is. The following program computes f:

1: $n := 0$;
2: **while** $f(n, x_0, \ldots, x_{s-1}) \neq 0$ **do**
3: $\quad n := n + 1$
4: **od**
5: $x_0 := n$

Thus program finds the first n such that $f(n, x_0, \ldots, x_{s-1}) = 0$. If no such n exists, the while loop does not terminate. If one $f(n, x_0, \ldots, x_{s-1})$ is undefined, the program does not terminate, too.

For the other direction, assume that we have a WHILE program $P =$ **while** $x_i \neq 0$ **do** P_1 **od**. The functions $u_\lambda(n, a_0, \ldots, a_\ell)$ constructed in the

proof of Lemma IV.4 is the content of the variable x_λ after executing P_1 n times. Thus

$$\mu u_\lambda(a_0, \ldots, a_\ell)$$

is the number of times the while loop is executed, and

$$u_\lambda(\mu u_\lambda(a_0, \ldots, a_\ell), a_0, \ldots, a_\ell)$$

is the content of x_λ after executing P. ■

Excursus: Programming systems V

We can also assign Gödel numbers to recursions schemes. We start by assignin numbers to the constant functions, projections, and successor function. For instance, $\langle 0, \langle s, c\rangle\rangle$ could stand for the function of arity s that has the value c everywhere, $\langle 1, \langle s, i\rangle\rangle$ encodes p_i^s and so on. Then we define Gödel numbers for composition and primitive and μ-recursion. If we have a functions f or arity s and s functions g_i of arity t and i and $j_1, \ldots, j_s$ are their Gödel numbers, then $\langle 3, t, s, i, j_1, \ldots, j_s\rangle$ is the Gödel number for their composition.

Let θ_i be the function that is computed by the recursion scheme with Gödel number i. If i is not a valid Gödel number, then θ_i is some dummy function, for instance, the function that is undefined everywhere. Then the sequence $(\theta_i)_{i\in\mathbb{N}}$ is a programming system.

It is universal, since we can use a simulation of the universal WHILE program to simulate recursion schemes. It is clearly acceptable, since composition is directly available in recursion schemes.

Part III

Complexity

19 Turing machines and complexity classes

Computability theory tries to separate problems that can be solved algorithmically from problems that cannot be solved algorithmically. Here, "can" and "cannot" means that there exists or does not exist a Turing machine, WHILE program, JAVA program, etc. that decides or recognizes the given language. We defined the classes REC and RE.

While it is nice to know that there is a Turing machine, WHILE program, or JAVA program that decides my problem, it does not help at all if the running time is so large that I will not live long enough to see the outcome. Complexity theory tries to separate problems that can be solved in an acceptable amount of time ("feasible" or "tractable" problems) from problems that cannot be solved in an acceptable amount of time ("infeasible" or "intracable" problems). Space comsumption is another resource that we will investigate. In contrast to the last part, we will use Turing machines as our main model of computation, which is traditionally used in complexity theory. The main reason is that one step of a Turing machine only manipulates a constant amount of data while a simple statement of a WHILE program can manipulate large numbers in one step, so the costs of a simple statement depend on the size of the operands. In the end, it will not matter at all, since usually, we will completely abstract from the concrete implemenation details and only give high level explanations of algorithms and reductions.

19.1 Deterministic complexity classes

Let M be a deterministic Turing machine and let x be an input. Assume that M halts on x, i.e., there is a unique accepting or rejecting configuration C_t such that $\mathrm{SC}(x) \vdash_M^* C_t$. Then, by definition, there is a unique sequence

$$\mathrm{SC}(x) \vdash_M C_1 \vdash_M \cdots \vdash_M C_t.$$

This sequence is called a *computation* of M on x. t is the *number of steps* that M performs on input x. We denote this number t by $\mathrm{Time}_M(x)$. If M does not halt on x, then the computation of M on x is infinite. In this case $\mathrm{Time}_M(x)$ is infinite.

For an $n \in \mathbb{N}$, we define the *time complexity* of M as

$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid |x| = n\}.$$

In other words, $\text{Time}_M(n)$ measures the *worst case*[1] behaviour of M on inputs of length n. Let $t : \mathbb{N} \to \mathbb{N}$ be some function. A deterministic Turing machine M is *t time bounded* if $\text{Time}_M(n) \leq t(n)$ for all n.

For a configuration $C = (q, (p_1, x_1), \ldots (p_k, x_k))$, $\text{Space}(C) = \max_{1 \leq \kappa \leq k} |x_\kappa|$ is the *space* used by the configuration. Occassionally, we will equip Turing machines with an *extra input tape*. This input tape contains, guess what, the input x of the Turing machine. This input tape is *read-only*, that is, the Turing machine can only read the symbols but not change them. (Technically, this is achieved by requiring that whenever the Turing machine reads a symbol on the input tape it has to write the same symbol.) What is an extra input tape good for? The space used on the input tape (that is, the symbols occupied by the input) is not counted in the definition of $\text{Space}(C)$. In this way, we can talk about sublinear space complexity.

Example 19.1 *Consider the language*

$$L = \{x \in \{0,1\}^* \mid \textit{the number of } 0\textit{'s in } x \textit{ equals the number of } 1\textit{'s}\}.$$

L can be recognized with space $\mathrm{O}(\log n)$. We read the input and for every 0 that we encounter, we increase a binary counter on the work tape by one. Then we read the input a second time and decrease the counter for every 1. We accept if in the end, the counter on the work tape is zero. In every step, we store number $\leq |x|$ on the work tape. This needs $\log |x|$ bits (on the work tape).

Let M be a deterministic Turing machine and let x be an input. First assume that M halts on x. Let

$$\text{SC}(x) \vdash_M C_1 \vdash_M \cdots \vdash_M C_t$$

be the computation of M on x. Then $\text{Space}_M(x) = \max\{\text{Space}(C_\tau) \mid 1 \leq \tau \leq t\}$. If M does not halt on x, then we build the maximum over infinitely many configurations. If the maximum does not exist, then $\text{Space}_M(x) = \infty$.

For an $n \in \mathbb{N}$, we define the *space complexity* of M as

$$\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid |x| = n\}.$$

In other words, $\text{Space}_M(n)$ measures the *worst case* behaviour of M on inputs of length n. Let $s : \mathbb{N} \to \mathbb{N}$ be some function. A deterministic Turing machine M is *s space bounded* if $\text{Space}_M(n) \leq s(n)$ for all n.

A language L is *deterministically t time decidable* iff there is a deterministic Turing machine M such that $L = L(M)$ and $\text{Time}_M(n) \leq t(n)$

[1] Worst case complexity has been a very fruitful concept in the design and analysis of algorithms. One could think of other ways of measuring complexity, for instance, average case complexiy. But if an algorithm is efficient in the worst case, it is also in the average case. And we avoid a discussion of what is a typical input.

for all n. In the same way, a function f is deterministically computable in time $t(n)$ iff there is a deterministic Turing machine M that computes f and $\text{Time}_M(n) \leq t(n)$ for all n. Note that a time bounded Turing machine always halts.

Definition 19.2 *Let $t : \mathbb{N} \to \mathbb{N}$. Then*

$$\begin{aligned}\mathsf{DTime}(t) &= \{L \mid L \textit{ is deterministically } t \textit{ time decidable}\},\\ \mathsf{DTime}_k(t) &= \{L \mid \textit{there is a } t \textit{ time bounded } k\textit{-tape Turing machine } M\\ &\qquad \textit{with } L = L(M)\}.\end{aligned}$$

For a set of functions T, $\mathsf{DTime}(T) = \bigcup_{t\in T} \mathsf{DTime}(t)$. $\mathsf{DTime}_k(T)$ is defined analogously.

The same is done for space complexity: A language L is *deterministically s space recognizable* if there is a deterministic Turing machine M such that $L = L(M)$ and $\text{Space}_M(n) \leq s(n)$ for all n. Note that a space bounded Turing machine might not halt on inputs that are not in $L(M)$. But we will see in the next chapters that one can effectively detect when a space bounded machine has entered an infinite loop. In the same way, a function f is deterministically computable in space s if there is a deterministic Turing machine M that computes f and $\text{Space}_M(n) \leq s(n)$ for all n. We will see that for space bounded computations, also sublinear functions s are meaningful. But to speak of sublinear space complexity, the input should not be counted. Therefore, we will use a Turing machine M with an extra input tape when considering space complexity.

Definition 19.3 *Let $s : \mathbb{N} \to \mathbb{N}$. Then*

$$\begin{aligned}\mathsf{DSpace}(s) &= \{L \mid L \textit{ is deterministically } s \textit{ space recognizable}\},\\ \mathsf{DSpace}_k(s) &= \{L \mid \textit{there is a } s \textit{ space bounded } k\textit{-tape Turing machine } M\\ &\qquad \textit{with } L = L(M)\}.\end{aligned}$$

In the definition of $\mathsf{DSpace}(s)$, the Turing machines have an additional input tape.

For a set of functions S, $\mathsf{DSpace}(S) = \bigcup_{s\in S} \mathsf{DSpace}(s)$. $\mathsf{DSpace}_k(S)$ is defined analogously.

Exercise 19.1 *Intuitively, it is clear that sublinear time is not very meaningful here.*[2] *Give a formal proof for this. In particular show: Let M be a deterministic Turing machine. Assume that there is an n such that M reads at most $n-1$ symbols of the input x for each x with $|x| = n$. Then there are words $a_1, \ldots, a_m$ with $|a_i| < n$ for all $1 \leq i \leq m$ such that $L(M) = \bigcup_{i=1}^{m} a_i\{0,1\}^*$.*

[2] This however changes if we have random access to the input and we are content with approximate results. Sublinear time algorithms are a very active area of research right now.

19.2 Nondeterministic complexity classes

In the following chapters, we will need *nondeterministic Turing machines*, too. Instead of a function

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k,$$

the transition function is now a function

$$\delta : Q \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^k \times \{L, S, R\}^k).$$

If in a given state q, a Turing machine reads the symbols $\gamma_1, \ldots, \gamma_k$ on the tapes, then it now has several possibilities of performing a step. Therefore, a configuration C now has several successor configurations, one for each possible step that the Turing machine can perform. This is very similar to nondeterministic finite automata.

Let M be a nondeterministic Turing machine and $x \in \Sigma^*$. We define a (possibly infinite) rooted labeled tree T, the *computation tree* of M on x as follows: The root is labeled with $\mathrm{SC}(x)$. As long as there is a node v that is labeled with a configuration C that is not a halting configuration, we do the following: Let $C_1, \ldots, C_\ell$ be all configurations such that $C \vdash_M C_\lambda$ for $1 \leq \lambda \leq \ell$. Now v gets ℓ children labeled with $C_1, \ldots, C_\ell$. (Note that it is possible that different nodes in T might have the same label.) A path from the root to a leaf in T is called a *computation path*. It is accepting if the configuration at the leaf is accepting, otherwise it is rejecting. There also might be infinite computation paths; these are considered to be rejecting. A nondeterministic Turing machine *accepts* an input x iff the corresponding computation tree has an accepting computation path. Note that if M is deterministic, then the computation tree is a path. A nondeterministic Turing machine recognizes the language

$$L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}.$$

Example 19.4 *Figure 19.1 shows a nondeterministic Turing machine M. In the state* ident*, M just goes to the right and leaves the content of the tape unchanged. In the state* invert*, it goes to the right and replaces every* 0 *by a* 1 *and vice versa. Whenever it reads a* 1*, M may nondeterministically choose to stay in its current state or to go the other state. M accepts if it is in the state* invert *after reading the whole input. Figure 19.2 shows the computation tree on input* 010*. It is quite easy to see that $L(M) = \{x \mid x$ contains at least one* $0\}$.

$\mathrm{Time}_M(x)$ is the length of a shortest accepting computation path in the computation tree of M on x, if such a path exists, and ∞ otherwise. We set

$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid |x| = n,\ x \in L(M)\}.$$

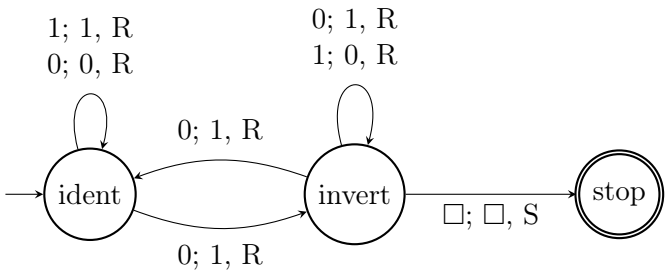


Figure 19.1: A nondeterministic 1-tape Turing machine. Whenever the Turing machine reads a 0, it may switch between the states ident and invert. In the state ident, the machine does not change the bits read. In the state invert, it exchanges 0's with 1's and vice versa.

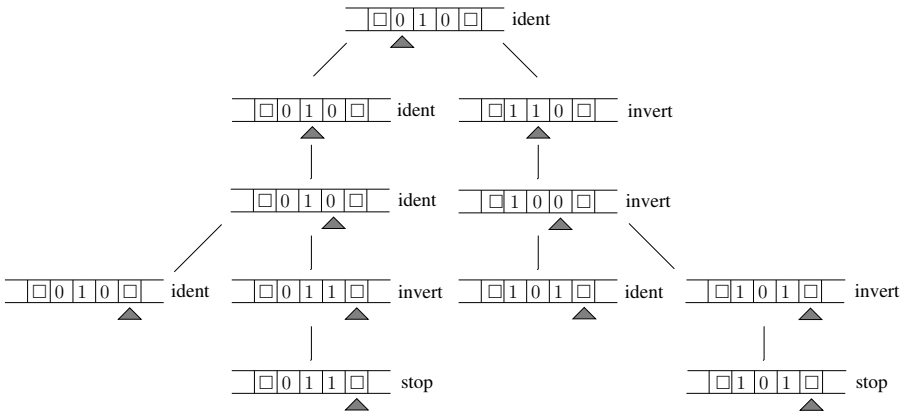


Figure 19.2: The computation tree of the Turing machine from Figure 19.1 on the word 010. The nodes are labeled with the configurations, right to the tape, there is the current state standing. The position of the head is marked by the small triangle. There are four paths in the computation tree, two of them are accepting (the state is stop).

If there is no $x \in L(M)$ with length n, then $\text{Time}_M(n) = 0$. Note that this definition is somewhat different to the deterministic case, where we took the maximum over *all* x of length n. Let $t : \mathbb{N} \to \mathbb{N}$ be some function. A nondeterministic Turing machine M is *weakly t time bounded* if $\text{Time}_M(n) \leq t(n)$ for all n.

Exercise 19.2 *Show that for any nondeterministic Turing machine M that is weakly t time bounded there is an equivalent Turing machine M' (i.e., $M(x) = M'(x)$ for all x) that is weakly $\mathrm{O}(t)$ time bounded such that for every input x, the computation tree of M' on x is a binary tree.*

If every computation of M on every x (and not only in $L(M)$) has length at most $t(|x|)$, then M is *strongly t time bounded*. Although strongly time bounded seems to be stronger than weakly time bounded, we will see soon that these two concepts lead to the same complexity classes for all "reasonable"[3] time bounds.

Definition 19.5 *Let $t : \mathbb{N} \to \mathbb{N}$. Then*

$$\begin{aligned}\mathsf{NTime}(t) &= \{L \mid \textit{there is a weakly } t \textit{ time bounded nondeterministic}\\ &\qquad \textit{Turing machine } M \textit{ with } L = L(M)\},\\ \mathsf{NTime}_k(t) &= \{L \mid \textit{there is a weakly } t \textit{ time bounded nondeterministic}\\ &\qquad k\textit{-tape Turing machine } M \textit{ with } L = L(M)\}.\end{aligned}$$

For a set of functions T, $\mathsf{NTime}(T) = \bigcup_{t \in T} \mathsf{NTime}(t)$. $\mathsf{NTime}_k(T)$ is defined analogously.

Warning! I am fully aware of the fact that there does not exists a physical realization of a nondeterministic Turing machine! (At least, I do not know of any.) Nondeterministic Turing machine are not interesting per se (at least not for an overwhelming majority of the world population), they are interesting because they characterize important classes of problems. The most important ones are the so-called NP-complete problems, a class which we will encounter soon. The example in Section 19.3 gives a first impression.

For a nondeterministic Turing machine M and an input $x \in L(M)$, we define space $\text{Space}_M(x)$ as follows: we take the minimum over all accepting paths of the maximum of the space used by any configuration along this path if such an accepting path exists, and ∞ otherwise. We set

$$\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid |x| = n,\ \ x \in L(M)\}.$$

If there is no x of length n in $L(M)$, then $\text{Space}_M(n) = 0$. Let $s : \mathbb{N} \to \mathbb{N}$ be some function. A nondeterministic Turing machine M is *weakly s space*

[3] One definition of reasonable is the following: Pick your favourite book on algorithms and open a random page. If you see a function $\mathbb{N} \to \mathbb{N}$ on this page, then it is reasonable, maybe except for the inverse of the Ackermann function.

bounded if $\text{Space}_M(n) \leq s(n)$ for all n. We define *strongly s space bounded* in the same way as we did for strongly time bounded.

Definition 19.6 *Let $s : \mathbb{N} \to \mathbb{N}$. Then*

$$\begin{aligned}\mathsf{NSpace}(s) = \{L \mid &\textit{there is a weakly } s \textit{ space bounded nondeterministic}\\ &\textit{Turing machine } M \textit{ with } L = L(M)\},\\ \mathsf{NSpace}_k(s) = \{L \mid &\textit{there is a weakly } s \textit{ space bounded nondeterministic}\\ &k\textit{-tape Turing machine } M \textit{ with } L = L(M)\}.\end{aligned}$$

In the case of $\mathsf{NSpace}(s)$*, the Turing machines have an extra input tape.*

19.3 An example

Consider the following arithmetic formula

$$x_1 + 2x_2(1 - x_1) + x_3.$$

We want to know whether we can assign the values 0 and 1 to the variables in such a way that the formula evaluates to 1. Above $x_1 \mapsto 1$, $x_2 \mapsto 0$, and $x_3 \mapsto 0$ is such an assignment. The formula

$$x_1(1 - x_1)$$

does not have such an assignment. We want to decide whether a given formula has such an assignment or not.

To make formulas accessible to Turing machines, we have to encode them as binary strings. The actual way how we do this will not matter in the following, as long as the encoding is "easily accessible". Here, this means that given an assignment, we can easily evaluate the formula F in time, say, $O(\ell^3)$ [4] where ℓ is the length of (the encoding of) the formula.

The following excursus formalizes the problem, but I recommend to skip it first.

Excursus: Formalization

Let $X = \{x_1, x_2, \dots\}$ be a set of variables. *Arithmetic formulas* are defined inductively:

1. Every $x \in X$ and every $z \in \mathbb{Z}$ is an arithmetic formula.

[4] $O(\ell^3)$ can be easily achieved for reasonable encodings: A formula F of length ℓ has at most ℓ arithmetic operations and the value of the formula in the end has at most ℓ bits (proof by induction). Addition and multiplication can be performed in time $O(\ell^2)$ by the methods that you learn in school and we have $\leq \ell$ of them. Using more sophisticated methods and a better analysis, one can bring down the evaluation time to $O(\ell^{1+\epsilon})$ for any $\epsilon > 0$.

2. If F and G are arithmetic formulas, then $(F \cdot G)$ and $(F + G)$ are arithmetic formulas.

An assignment is a map $a : X \to \mathbb{Z}$. If we replace every occurence of a variable x by $a(x)$ in a formula F, then F just describes an integer. We extend a to the set of all arithmetic formulas inductively along the above definition:

1. $a(z) = z$ for all $z \in \mathbb{Z}$.
2. $a(F \cdot G) = a(F) \cdot a(G)$ and $a(F + G) = a(F) + a(G)$ for formulas F and G.

Since in every formula, only a finite number of variable occur, we usually restrict assignments to the variables occurring in a given formula. An assignment is called an S assignment for some $S \subseteq \mathbb{Z}$, if $\operatorname{im} a \subseteq S$.

We can encode arithmetic formulas as follows: For instance, we can encode the variable x_i by $[0, \operatorname{bin}(i)]$ and a constant z by $[1, \sigma(z), \operatorname{bin}(|z|)]$ where $\sigma(z)$ is 1 if $z \geq 0$ and 0 otherwise. Then we define the encoding c inductively by $c(F \cdot G) = [10, c(F), c(G)]$ and $c(F + G) = [11, c(F), c(G)]$, where $[\dots]$ is our pairing function. This is a very structured encoding, since it explicitly stores the order in which operations are performed. Alternatively, we first encode x_i by the string $x \operatorname{bin}(i)$ and z by $\sigma(z) \operatorname{bin}(|z|)$. Now we can view our formula as a string over the alphabet $\{(,), +, \cdot, x, 0, 1\}$. To get a string over $\{0, 1\}$, we just replace each of the seven symbols by a different binary string of fixed length. (Three is sufficient, since $2^3 \geq 7$.) This is a rather unstructured encoding. Nevertheless, both encodings allow us to evaluate the formula in time $\mathrm{O}(\ell^3)$.

Since the encoding does not matter in the following, we will not specify it explicitly. We just assume that the encoding is reasonable. Since there is usually no danger of confusion, we will even write F for both the formula itself (as a mathematical object) and its encoding (instead of $c(F)$ or something like that). Let

$$\mathrm{AFSAT} = \{F \mid \text{there is an } \{0,1\} \text{ assignment such that } F \text{ evaluates to } 1.\}$$

How hard is it to decide whether a given formula F has a $\{0, 1\}$ assignment such that F evaluates to 1? I.e., how hard is it to decide whether $F \in \mathrm{AFSAT}$? Assume that F has length n. Since F can have at most n variables, there are 2^n possible assignments. For each of them we can check in time $\mathrm{O}(n^3)$ whether F evaluates to 1 or not. Thus

$$\mathrm{AFSAT} \in \mathsf{DTime}(\mathrm{O}(2^n \cdot n^3))$$

A nondeterministic Turing machine can do the following. It first reads the input and for each symbol it reads it has two options: Either write a 0 on the second tape or a 1, see Figure 19.3. The computation tree has 2^n paths. At every leaf, the machine has written one string from $\{0, 1\}^n$ on the second tape. We now interpret this string as an assignment to the variables, ignoring some of the bits if there are fewer than n variables. The machine now just (deterministically) evaluates the formula with respect to this assignment and accepts if the outcome is 1 and rejects otherwise.

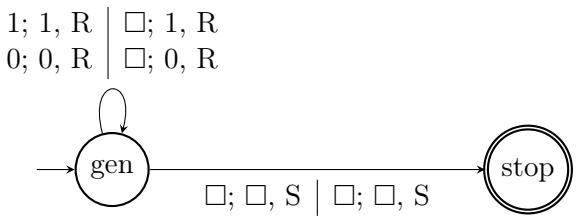


Figure 19.3: A nondeterminstic Turing machine that generates all strings from $\{0,1\}^n$.

The machine is clearly $O(n^3)$ time bounded, since the length of each computation path is dominated by the time needed for evaluating the formula. It correctly decides whether $F \in$ AFSAT, too: If there is a $\{0,1\}$ assignment such that F evaluates to 1, then it will be generated along some computation path and at the end of this path, the Turing machine will accept. If there is no such assignment, then there cannot be any accepting computation path at all. Hence

$$\text{AFSAT} \in \mathsf{NTime}(O(n^3)).$$

The deterministic Turing machine tries all possible assignments, one after another. It is an open problem whether there is a substantially more clever way. A nondeterministic Turing machine can try them in parallel. Or we can view it like this: If we are given an assignment, then we can easily check whether F evaluates to 1 under *this* assignment.

Like AFSAT (which is more of pedagogical value), there are an abundance of similar and very, very, very important problems (so called NP-complete problems), that have the same property: To find a solution, we do not know anything really better than trying all possible solution. But if we get a potential solution, we can easily verify whether it is really a solution. This is what gives nondeterminism its right to exists ... (and there might be some other reasons, too)

Excursus: Complexity of WHILE programs

We proved that WHILE computable functions and Turing computable functions are essentially the same. But what about time and space complexity. Are there functions that can be computed much faster by WHILE programs than be Turing machines or vice versa? The answers are "no" and "it depends".

First of all, Turing machines get bit strings as inputs and WHILE programs get natural numbers. The complexity of a Turing machine is measured in the length of the input. So we should measure the running time of a WHILE program as a function in $|\operatorname{cod}^{-1}(n)| = \log n$. The running time of a WHILE program on some input i is the number of simple statements that are executed on i. (If a simple statement is executed several times, then it is of course counted this many times.) The running time on inputs of length ℓ is the maximum over the running times on

inputs i with $|\operatorname{cod}^{-1}(i)| = \ell$. If $S = (\sigma_0, \ldots, \sigma_\ell)$ is a state of a WHILE program, then the space used by this state is $\operatorname{Space}(S) := \max\{|\operatorname{cod}^{-1}(\sigma_\lambda)| \mid 0 \leq \lambda \leq \ell\}$.

One has to be careful when defining the running time of a WHILE program. We here simply count the number of simple statements. This is okay here, since every simple statement can enlarge the space used by a state by a constant amount. If multiplication was also a simple operation, then we could double the space with one simple operation. This would make WHILE programs very powerful. To get meaningful results in this case, a simple operation has to cost $\operatorname{Space}(S)$ where S is the current state.

Now let P be a simple statement. Assume we have a state $S = (\sigma_0, \ldots, \sigma_\ell)$ and let $S' = (\sigma'_0, \ldots \sigma'_\ell) := \Phi_P(S)$. We claim that $\operatorname{Space}(S') \leq 1 + \operatorname{Space}(S)$, if $\operatorname{Space}(S)$ is large enough. Assume that $P = x_i := x_j + x_k$. Then $\sigma'_i = \sigma_j + \sigma_k$ and all other entries are not changed. But then $|\operatorname{cod}^{-1}(\sigma'_i)| \leq 1 + \max\{|\operatorname{cod}^{-1}(\sigma_j)|, |\operatorname{cod}^{-1}(\sigma_k)|\}$. The same is true if P is a subtraction. The case $P = x_i := c$ can only change $\operatorname{Space}(S)$ if $\operatorname{Space}(S) \leq |\operatorname{cod}^{-1}(c)|$. But every WHILE program contains only a finite number of constants. This means the assigning constants does not have any asymptotic effect. It follows by induction that by executing t simple statements, we can increase the space consumption by at most t.

Now consider the simulation of a WHILE program by a Turing machine. We first replaced the WHILE program by a GOTO programm but the number of simple statements and the number of space used by the GOTO program is the essentially same. In this simulation, the content of each variable is stored on a different tape. Hence we do not need more space in this simulation than the GOTO program does. To simulate a simple statement, we have just to add or subtract two numbers. This can be done in time linear in the size of the operands. (We only used incrementation and decrementation in our simulation, but it is easily extended to addition and subtraction.) Thus we only get a quadratic slowdown when simulating WHILE programs by Turing machines.

When we simulated Turing machines by WHILE programs, we stored the tapes in array and then could easily do a step-by-step simulation. So the running time of the Turing machines is multiplied by the time needed to manipulate the arrays. If we use the ordinary pairing function $\langle ., . \rangle$, then the sizes can explode. But since we only store elements $a_0, \ldots, a_s$ from a finite set $\{0, 1, \ldots, b\}$, say, we can do this by interpreting $a_0, \ldots, a_s$ as the digits of a b-nary number $\sum_{i=0}^{s} a_i b^i$. In this way, we again only get a quadratic slowdown.

20 Tape reduction, compression, and acceleration

In this chapter, we further investigate the Turing machine model.

20.1 Tape reduction

Definition 20.1 *A deterministic Turing machine M* simulates *a Turing machine M', if $L(M) = L(M')$ and for all inputs x, M halts on x iff M' halts on x.*

Theorem 20.2 *Every deterministic Turing machine can be simulated by a deterministic 1-tape Turing machine.*

Proof. Let M a be k-tape Turing machine. We construct a 1-tape Turing machine S that simulates M. S simulates one step of M by a sequence of steps.

We have to store the content of all k tapes on one tape. We think of the tape of S divided into $2k$ tracks. To this aim, we enlarge the tape alphabet of S. The work alphabet of S is $\Gamma' = (\Gamma \times \{*, -\})^k \cup \Sigma \cup \{\Box\}$. The $(2\kappa-1)$th component of a symbol of Γ' stores the content of the κth tape of M. The 2κth component is used to mark the position of the head on the κth tape. There will be exactly one $*$ on the 2κth track, this $*$ will mark the position of the head. All other entries of the track will be filled with $-$'s. Figure 20.1 depicts this construction: One column on the righthand side of the figure is one symbol on the tape of S. In particular, a $\Box$ or $-$ in such a column is *not* the blank of S. The blank of S is just $\Box$, one column just filled with one $\Box$.

Figure 20.2 shows how the simulating machine S works. Let $x \in \Sigma^*$ be the given input. First, S replaces the input x by the corresponding symbols from $(\Gamma \times \{*, -\})^k$. The first track contains x, all other odd tracks contain blanks. On the even tracks, the $*$ is in the first position of each track.

One step of M is now simulated as follows: S always starts on the left-most position of the tape visited so far and moves to the right until it reaches the first blank (of S)[1]. On its way, S collects the k symbols under the heads of M and stores them in its finite control. Once S has collected all the symbols, it can simulate the transition of M. It changes the state accordingly and now moves to the left until it reaches the first blank (of S). On its way back,

[1] A blank of S indicates that we reached a position that M has not visited so far on any of its tapes.

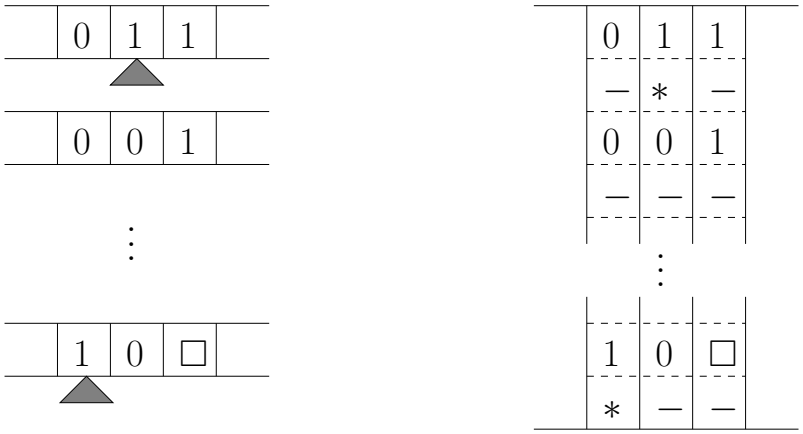


Figure 20.1: Lefthand side: The k tapes of the k-tape Turing machine M. Righthand-side: The one and only tape of the simulating machine S. The tape of S is divided into $2k$ tracks, two for each tape of M. The first track of each such pair of tracks stores the content of the corresponding tape of M, the second stores the position of the head which is marked by "$*$".

it makes the changes that M would make. It replaces the entries in the components marked by a $*$ and moves the $*$ in the corresponding direction.

If M has not halted yet, S repeats the loop described above. If M halts, S halts, too, and accepts iff M accepts. ∎

Remark 20.3 *The above construction also works for nondeterministic Turing machines. Whenever S has collected all the symbols and simulates the actual transition of M, it chooses one possible transition nondeterministically.*

Remark 20.4 *If M has an additional input tape, then we can also equip S with an additional input tape. If M has a sublinear space bound, then also S has a sublinear space bound.*

Remark 20.5 (Implementation details) *The description of the simulator in the above proof is rather high level. A more low level description, i.e., the explicit transition function, usually does not provide any insights. But we tacitely assume that you can write down—at least in principle—the transition function of the 1-tape Turing machine constructed above.*

Let's convince ourselves that we can really do this: Consider the part of S that collects the symbols that M would read. The states are of the form $\{\text{collect}\} \times Q \times (\Gamma \cup \{/\})^k$. The first entry of a tuple $(\text{collect}, q, \gamma_1, \ldots, \gamma_k)$ indicates that we are in a collection phase. (If the collection phase were the only phase that uses tuple of the form $Q \times (\Gamma \cup \{/\})^k$, then we could skip

Input: $x \in \Sigma^*$
Output: accept if $x \in L(M)$, reject otherwise

1. S replaces the input x by the corresponding symbol of Γ', i.e., x_1 is replaced by $(x_1, *, \square, *, \ldots, \square, *)$ and each other x_ν is replaced by $(x_\nu, -, \square, -, \ldots, \square, -)$.
2. S always stores the current state of M in its finite control. In the beginning, this is the starting state of M.
3. As long as M does not halt, S repeats the following:
   (a) S moves to the right until it reaches the first blank. On its way to the right, S reads the symbols that the heads of M are reading and stores them in its finite control.
   (b) When S reaches the right end of the tape content, it has gathered all the information to simulate one step of M. It changes the internally stored state of M accordingly.
   (c) S now moves to the left until it reaches the first blank. On its way to the left, S replaces the entries in components that are marked by a $*$ by the symbol that M would write on the corresponding tape and moves the $*$ like M would move the corresponding head.
   If S has to move one of the markers $*$ to a cell that still contains $\square$, the blank of S, then it first replaces this blank by $(\square, -, \ldots, \square, -)$.
4. If M accepts, S accepts. Otherwise, S rejects.

Figure 20.2: The simulator S.

this first component.) The second component stores the current state of M. It shall not be changed during the collection phase. Finally, γ_κ stores the symbol that is read by M on the κth tape. $\gamma_\kappa = /$ indicates that the position of the head on tape κ has not been found yet.

The transition function δ' of S (restricted to the states of the collect phase) is now defined as follows:

$$\delta'((\text{collect}, q, \gamma_1, \ldots, \gamma_k), (\eta_1, \ldots, \eta_{2k})) = ((\text{collect}, q, \gamma'_1, \ldots, \gamma'_k), (\eta_1, \ldots, \eta_{2k}), R)$$

for all $q \in Q, \gamma_1, \ldots, \gamma_k \in \Gamma \cup \{/\}$ where

$$\gamma'_\kappa = \begin{cases} \gamma_\kappa & \text{if } \eta_{2\kappa} = - \\ \eta_{2\kappa-1} & \text{if } \eta_{2\kappa} = * \end{cases}$$

for $1 \leq \kappa \leq k$. If the symbol $\eta_{2\kappa}$ on the 2κth track is $*$, then we found the head on the κth tape and store $\eta_{2\kappa-1}$, the symbol that M reads on the κth tape, in the state of S.

Definition 20.6 *Let $t, s : \mathbb{N} \to \mathbb{N}$. Then*

$$\mathsf{DTimeSpace}(t, s) = \{L \mid \text{there is a } t \text{ time and } s \text{ space bounded Turing machine } M \text{ with } L = L(M)\}.$$

$\mathsf{DTimeSpace}_k(t, s)$ *and* $\mathsf{NTimeSpace}_k(t, s), \ldots$ *are defined accordingly.*

Theorem 20.7 *For all $t, s : \mathbb{N} \to \mathbb{N}$,*

$$\mathsf{DTimeSpace}(t, s) \subseteq \mathsf{DTimeSpace}_1(\mathrm{O}(ts), \mathrm{O}(s)),$$
$$\mathsf{NTimeSpace}(t, s) \subseteq \mathsf{NTimeSpace}_1(\mathrm{O}(ts), \mathrm{O}(s)).$$

If $s(n) = \mathrm{O}(n)$, then the one-tape Turing machine needs an extra input tape.

Proof. Let $L \in \mathsf{DTimeSpace}(t, s)$. Let M be a deterministic Turing machine with $L = L(M)$. Assume that M has k tapes. The Turing machine S in Theorem 20.2 simulates one step of M by $\mathrm{O}(s(|x|))$ steps (where x is the given input). M makes at most $t(|x|)$ steps. Furthermore, S does not use more twice the space M uses. (On each track, S does not use more space then M on the corresponding tape. But on one tape M could use the cells to the left of cell 0 and on the other to the right.)

The nondeterministic case follows in the same way by Remark 20.3. ∎

Corollary 20.8 *For all $t : \mathbb{N} \to \mathbb{N}$,*

$$\mathsf{DTime}(t) \subseteq \mathsf{DTime}_1(\mathrm{O}(t^2)),$$
$$\mathsf{NTime}(t) \subseteq \mathsf{NTime}_1(\mathrm{O}(t^2)).$$

Proof. Let M be a t time bounded Turing machine. In t steps, a turing machine can visit at most t cells. Thus $\mathrm{Space}_M(n) \leq t(n)$ for all n, and the corollary follows from Theorem 20.7. ∎

20.2 Tape compression

The aim of this and the next section is to show that we do not need to take care of constant factors when measuring space and time. Every Turing machine M can be simulated by another one that uses only a constant fraction of the space used by M. (Note that we will not even care about polynomial factors in the following.)

Theorem 20.9 *For all $0 < \epsilon \leq 1$ and all $s : \mathbb{N} \to \mathbb{N}$,*

$$\mathsf{DSpace}(s(n)) \subseteq \mathsf{DSpace}_1(\lceil \epsilon s(n) \rceil),$$
$$\mathsf{NSpace}(s(n)) \subseteq \mathsf{NSpace}_1(\lceil \epsilon s(n) \rceil).$$

If $s(n) = O(n)$, then the one-tape Turing machines need an extra input tape.

Proof overview: In the same way as a 64 bit architectur can store more information in one memory cell than an 8 bit architectur, we enlarge the tape alphabet to store several symbols in one symbol and then just simulate.

Proof. Let $c = \lceil 1/\epsilon \rceil$. Let M be a deterministic k-tape Turing machine with work alphabet Γ. We simulate M by a deterministic k-tape Turing machine with work alphabet $\Gamma' = \Gamma^c \cup \Sigma \cup \{\square\}$. A block of c contiguous cells of a tape of M are coded into one cell of S. Instead of s cells, S then uses only $\lceil s/c \rceil \leq \lceil \epsilon s \rceil$ cells. S can simulate M step by step. S stores the position of the head within a block of c cells in its state. If M moves his head within such a block, then S does not move its head at all but just changes the symbol.

If M does not have an extra input tape, then it first has to compress the input of length n into $\lceil n/c \rceil$ cells. But in this case, M can never use less than space n. If M has an extra input tape, this step is not necessary.

If M is nondeterministic, the same simulation works. ∎

Remark 20.10 (Implementation details) *Again, let's try to formalize a part of the transition function δ' of S. The states of S are of the form $Q \times \{1, \ldots, c\}$. (q, i) means that M is in state q and its head is on the ith symbol of the current block. Assume that $\delta(q, \eta) = (q', \eta', R)$. Then*

$$\delta'((q,i),(\gamma_1,\ldots,\gamma_c)) = \begin{cases} ((q',i+1),(\gamma'_1,\ldots,\gamma'_c),S) & \text{if } i < c \\ ((q',1),(\gamma'_1,\ldots,\gamma'_c),R) & \text{if } i = c \end{cases}$$

for all $q \in Q$, $i \in \{1, \ldots, c\}$, and all $(\gamma_1, \ldots, \gamma_c)$ with $\gamma_i = \eta$, where $\gamma'_j = \gamma_j$ for $j \neq i$ and $\gamma'_i = \eta'$.

Exercise 20.1 *Show the following "converse" of Theorem 20.9: For any s space and t time bounded Turing machine M with input alphabet $\{0, 1\}$, there is a $\mathrm{O}(s)$ space and $\mathrm{O}(t)$ time bounded Turing machine that only uses the work alphabet $\{0, 1, \square\}$.*

20.3 Acceleration

Next, we prove a similar speed up for time. This simulation is a little more complicated than the previous one.

Exercise 20.2 *Show the following: For all $k \geq 2$, all $t : \mathbb{N} \to \mathbb{N}$, and all $0 < \epsilon \leq 1$,*

$$\mathsf{DTime}_k(t(n)) \subseteq \mathsf{DTime}_k(n + \epsilon(n + t(n)))$$
$$\mathsf{NTime}_k(t(n)) \subseteq \mathsf{NTime}_k(n + \epsilon(n + t(n))).$$

Proof overview: Like in Theorem 20.9, we want to store several, say c, cells into one. To get a speed up, the simulating machine S now has to simulate c steps in one step. This is no problem if M stays within the c cells of one block, since we just can precompute the outcome. Problematic is the following case: During the c steps, the Turing machine M goes back and forth between two cells that belong to different (but neighboured) blocks. To overcome this problem, S always stores three blocks in its finite control: The block B where the head of M is located and the blocks to the left and the right of B. With these three blocks, S can simulate c steps of M in its finite control. Then S updates the tape content. If M left the block B, then S also has to update the blocks in its finite control.

If $t(n) = \omega(n)$, then we can speed the computation by any factor ϵ in Exercise 20.2. If $t(n) = \mathrm{O}(n)$, then we can get a running time of $(1+\epsilon)n$ for any $\epsilon > 0$.

What to measure?

Time and space consumption of Turing machines should only be measured up to constant factors!

20.4 Further exercises

Exercise 20.3 *Prove the following. Let c be some constant. Every k-tape Turing machine M can be simulated by a k-tape Turing machine S such that*

$$\mathrm{Time}_S(n) = \begin{cases} n & \text{if } n \leq c \\ \mathrm{Time}_M(n) + c & \text{otherwise} \end{cases}$$
$$\mathrm{Space}_S(n) = \begin{cases} 0 & \text{if } n \leq c \\ \mathrm{Space}_M(n) & \text{otherwise} \end{cases}$$

In other words, only the asymptotic behaviour matters.

21 Space versus Time, Nondeterminism versus Determinism

21.1 Constructible functions

Definition 21.1 *Let $s, t : \mathbb{N} \to \mathbb{N}$.*

1. *t is* time constructible *if there is a $\mathrm{O}(t)$ time bounded deterministic Turing machine M that computes the function $1^n \mapsto \mathrm{bin}(t(n))$.*

2. *s is* space constructible *if there is an $\mathrm{O}(s)$ space bounded deterministic Turing machine M (with extra input tape) that computes the function $1^n \mapsto \mathrm{bin}(s(n))$.*

Above, $\mathrm{bin}(n)$ denotes the binary representation of n.

Exercise 21.1 *Show the following:*

1. *If t is time constructible, then there is a $\mathrm{O}(t)$ time bounded deterministic Turing machine that on input x writes $1^{t(|x|)}$ on one of its tapes.*

2. *If s is space constructible, then there is a s space bounded deterministic Turing machine (with extra input tape) that on input x writes $1^{s(|x|)}$ on one of its tapes.*

Time and space constructible functions "behave well". One examples for this is the following result.

Lemma 21.2 *Let t be time constructible and s be space constructible.*

1. *If $L \in \mathsf{NTime}(t)$ then there is a strongly $\mathrm{O}(t)$ time bounded nondeterministic Turing machine N with $L = L(N)$.*

2. *If $L \in \mathsf{NSpace}(s)$, then there is a strongly $\mathrm{O}(s)$ space bounded nondetermistic Turing machine N with $L = L(N)$.*

Proof. We start with the first statement: Let M be some weakly t time bounded Turing machine with $L(M) = L$. Consider the following turing machine N:

Input: x

1. Construct $\text{bin}(t(|x|))$ on some extra tapes.
2. Simulate M step by step.
   On the extra tape, count the number of simulated steps with a binary counter.
3. When more than $t(|x|)$ steps have been simulated, then stop and reject.
4. If M halts earlier, then accept if M has accepted and reject otherwise.

N is clearly $\text{O}(t)$ time bounded, since counting to $t(|x|)$ in binary can be done in $\text{O}(t(|x|))$ time (amortized analysis!). If M accepts x, then there is an accepting path whose length is at most $t(|x|)$. This path will be simulated by N and hence N will accept x. If M does not accept x, then all paths in the computation tree are either infinite or rejecting. In both cases, the corresponding path of N will be rejecting.

For the second part, let M be some weakly s space bounded Turing machine with an extra input tape such that $L(M') = L$. Consider the following Turing machine N:

Input: x

1. Mark $2s(|x|)$ cells with a new symbol ■ on each work tape (see Exercise 21.1), $s(|x|)$ to the left of cell 0 and $s(|x|)$ to the right.
2. Simulate M on x pretending that each ■ is a □.
3. When we read a real blank □ during the simulation, then we stop and reject.

N is clearly $\text{O}(s)$ space bounded. If M accepts x, then there is an accepting computation path on which M is s space bounded. When N simulates this path, then N will never reach a □ and hence will accept. If M does not accept x, then N will not accept, too. ■

Most interesting functions are space and time constructible.

Exercise 21.2 *Let $a, b, c \in \mathbb{N}$ and let $f(n) = 2^{an} \cdot n^b \cdot \log^c(n)$.*

1. *If $f \in \omega(n)$, then f is time constructible.*

2. *f is also space constructible. This even holds if a, b, c are rational numberers provided that $f \in \Omega(\log n)$.*

21.2 The configuration graph

Let M be a Turing machine. The set of all configurations of M together with the relation $\vdash_M$ can be interpreted as an infinite directed graph. The node set of this graph is the set of all configurations. We denote this set by Conf_M. We denote this graph by $\mathrm{CG}_M = (\mathrm{Conf}_M, \vdash_M)$.[1] This is an infinite graph. But to decide whether a Turing machine accepts an input x, we just have to find out whether we can reach an accepting configuration from the starting configuration $\mathrm{SC}_M(x)$. This task is undecidable in general (it is the halting problem), but it becomes feasible when the Turing machine is time or space bounded. For a given space bound s, the relevant part of CG_M is finite.

Lemma 21.3 *Let M be an s space bounded Turing machine with $s(n) \geq \log n$ for all n. There is a constant c (depending on M) such that M on input x can reach at most $c^{s(|x|)}$ configurations from $\mathrm{SC}(x)$.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_{\mathrm{acc}})$ be a k-tape Turing machine. A configuration of M is described by the current state, the content of the work tapes and the position of the heads. There are $|Q|$ states, $|\Gamma|^{s(|x|)}$ possible contents of a tape and $s(|x|)$ possible positions of the heads. Thus the number of configurations is at most

$$|Q| \cdot \left(|\Gamma|^{s(|x|)}\right)^k \cdot s(|x|)^k \cdot (|x| + 2) \tag{21.1}$$

If the Turing machine does not have an extra input tape, the last factor $|x| + 2$ is not necessary. It is easy to see that (21.1) is bounded by $c^{s(|x|)}$ for some constant c only depending on $|Q|$, $|\Gamma|$, and k. (To bound the last factor $|x| + 2$, we need the assumption $s(n) \geq \log n$.) ∎

Exercise 21.3 *Give an upper bound for the constant c above.*

Corollary 21.4 *Let $s(n) \geq \log n$ for all n. If a deterministic s space bounded Turing machine halts on an input x, then it can perform at most $c^{s(|x|)}$ steps on x, where c is the constant from Lemma 21.3*

[1] Note that the binary relation $\vdash_M$ is nothing else than a set of pairs of elements from Conf_M, that is, directed edges.

Proof. By contradiction: If M makes more steps, then the computation of M would contain the same configuration twice by the pigeon hole principle. Since M is deterministic, this means that the computation is infinite. ∎

Corollary 21.5 *Let $s(n) \geq \log n$ be space constructible. Then $\mathsf{DSpace}(s)$ is closed under complement, i.e., if $L \in \mathsf{DSpace}(s)$ so is $\bar{L}$.*

Proof. Let M be an s space bounded deterministic Turing machine for L. We construct a deterministic Turing machine $\bar{M}$ as follows: $\bar{M}$ simulates M step by step. If M halts, then $\bar{M}$ halts, too, and accepts if M rejects and vice versa. Problematic is the case when M does not halt. Here $\bar{M}$ has to halt and accept. M marks $s(n)$ cells on an extra tape before starting the simulation and uses it as a c-nary counter to count the number of steps of M. If M makes more than $c^{s(n)}$ steps, then $\bar{M}$ stops and accepts. ∎

Remark 21.6 *Corollary 21.5 is trivially true for deterministic* time *classes, we just have to exchange accepting states with rejecting states. Whether nondeterministic time classes are closed under complementation is a major open problem. Nondeteterministic space classes (with constructible space bounds) are closed under complementation. This is the remarkable Immerman-Szelepcsényi theorem. Note that simply exchanging accepting with rejecting states does not work.*

Exercise 21.4 *Show the following: Let M be a nondeterministic Turing machine that is weakly s space bounded. If M accepts some input x, then there is an accepting path in the computation tree of M on x that has length at most $c^{s(|x|)}$.*

We will use the configuration graph to simulate space bounded Turing machines by time bounded ones and nondeterministic ones by deterministic ones. The following observation is crucial for the proof.

Observation 21.7 *Let G be a graph. If there is a path of length ℓ from u to v, then there is a node w such that there are paths from u to w and w to v of length $\lceil \ell/2 \rceil$ and $\lfloor \ell/2 \rfloor$, respectively.*

Lemma 21.8 *Let M be an s space bounded (deterministic or nondeterministic) Turing machine where $s(n) \geq \log n$ is space constructible. There is a $2^{\mathrm{O}(s(|x|))}$ time bounded deterministic Turing machine M_1 and $\mathrm{O}(s^2(|x|))$ space bounded deterministic Turing machine M_2 that given x decides whether an accepting configuration in CG_M is reachable from $\mathrm{SC}_M(x)$.*

Proof. To achieve the time bound $2^{\mathrm{O}(s(|x|))}$, M_1 simply generates the whole graph of all configurations that use space $s(|x|)$. We can enumerate

these configurations, because s is space constructible.[2] We can enumerate all edges in $\vdash_M$, since to find the successor of a configuration, we need time $O(\max\{|x|, s(|x|)\})$. Since $2^{s(|x|)} \geq |x|$, this is within the required time bound. By Lemma 21.3, it is now sufficient to check whether M_1 could reach an accepting configuration from $\mathrm{SC}(x)$. This can be done by your favourite connectivity algorithm.

The real fun is to do this in space $O(s^2)$, i.e., to construct M_2. To achieve this, we define

$$R(C, C', \ell) = \begin{cases} 1 & \text{if } C' \text{ can be reached from } C \text{ with } \leq \ell \text{ steps,} \\ 0 & \text{otherwise.} \end{cases}$$

If we can compute $R(C, C', \ell)$ for every configuration C and C' and $\ell \leq c^{s(n)}$ in space $O(s^2(|x|))$, then we are done. Here c is the constant from Lemma 21.3. We enumerate all accepting configurations C—there are at most $c^{s(|n|)}$—and compute $R(\mathrm{SC}(x), C, c^{s(n)})$. We accept iff at least one of these values is one. Note that we can reuse the space when computing $R(\mathrm{SC}(x), C, c^{s(n)})$ for the next C.

We will compute $R(C, C', \ell)$ recursively. We use the identity

$$\begin{aligned} R(C, C', \ell) = 1 \iff\ & \text{there is a configuration } C'' \text{ such that} \\ & R(C, C'', \lceil \ell/2 \rceil) = R(C'', C', \lfloor \ell/2 \rfloor) = 1 \\ & \text{or } C \vdash_M C'. \end{aligned}$$

This suggests the following recursive approach. Enumerate all configurations C'', one at a time. Then compute $R(C, C'', \lceil \ell/2 \rceil)$. If this value is 1, then also compute $R(C'', C', \lfloor \ell/2 \rfloor)$. If this is 1, then we are done. If one of the two values is zero, then we try the next C''. If we tried all C'' without success, then M_2 rejects.

Let $S(\ell)$ denote the maximum space needed to compute $R(C, C'', \ell)$ for any C, C''. We have

$$\begin{aligned} S(\ell) &\leq O(s(|x|)) + S(\lceil \ell/2 \rceil), \\ S(1) &\leq O(s(|x|)). \end{aligned}$$

To see this note that we need $O(s(|x|))$ space to write down C'' and then we need $S(\lceil \ell/2 \rceil)$ space to compute $R(C, C'', \lceil \ell/2 \rceil)$ and $R(C'', C', \lceil \ell/2 \rceil)$, since we can use the same cells twice. Therefore $S(\ell) = O(s(|x|) \cdot \log \ell)$. In particular, $S(c^{s(|x|)}) = O(s^2(|x|))$. ∎

[2] To enumerate the configurations, we have to encode them as strings. This can be done by giving the states numbers and writing all numbers in the configuration down in binary. We separate the components of a configuration by using some new symbol. In this way, the strings have length $\leq c \cdot s(|x|)$ for some constant c. We can now enumerate all strings of length $c \cdot s(|x|)$ for instance in lexicographical order and check whether it is a valid configuration.

21.3 Space versus time

As a first application, we show that a space bounded Turing machines can be simulated by time bounded ones with an exponential loss.

Theorem 21.9 *Let $s(n) \geq \log(n)$ be space constructible. Then*

$$\mathsf{DSpace}(s) \subseteq \mathsf{NSpace}(s) \subseteq \mathsf{DTime}(2^{\mathrm{O}(s)}).$$

Proof. The first inclusion is trivial. The second inclusion follows from Lemma 21.8. ■

21.4 Nondeterminism versus determinism

Theorem 21.10 *Let t be time constructible. Then*

$$\mathsf{NTime}(t) \subseteq \mathsf{NSpace}(t) \subseteq \mathsf{DTime}(2^{\mathrm{O}(t)}).$$

Proof. The first inclusion is trivial, since a t time bounded Turing machine can use at most $t(|x|)$ cells. The second inclusion follows from Lemma 21.8. ■

Theorem 21.11 (Savitch) *Let $s(n) \geq \log n$ be space constructible. Then*

$$\mathsf{NSpace}(s) \subseteq \mathsf{DSpace}(\mathrm{O}(s^2)).$$

Proof. Again, this follows from Lemma 21.8. ■

22 P and NP

We are looking for complexity classes, that are *robust* in the sense that "reasonable" changes to the machine model should not change the class. Furthermore, the classes should also characterize interesting problems.

Definition 22.1

$$\mathsf{P} = \bigcup_{i \in \mathbb{N}} \mathsf{DTime}(\mathrm{O}(n^i))$$
$$\mathsf{NP} = \bigcup_{i \in \mathbb{N}} \mathsf{NTime}(\mathrm{O}(n^i))$$

P (P stands for *polynomial time*) is *the* class of problems that are considered to be feasible or tractable. Frankly, an algorithm with running time $\mathrm{O}(n^{1024})$ is not feasible in practice, but the definition above has been very fruitful. If a natural problem turns out to be in P, then we usually will have an algorithm whose running time has a low exponent. In this sense, P contains all languages that we *can* decide quickly.

NP (NP stands for *nondeterministic polynomial time* and *not* for non-polynomial time) on the other hand, is a class of languages that we *would like to* decide quickly. There are thousands of interesting and important problems in NP for which we do not know deterministic polynomial time algorithms.

The class P is a robust class. A language that can be decided by a deterministic Turing machine in polynomial time can be decided by a WHILE program in polynomial time and vice versa. (This follows easily by inspecting the simulations that we designed in the first part of the lecture. But read the excursus in Chapter 19.) This is also true for NP, if we equip WHILE programs with nondeterminism in a suitable way.

The question whether $\mathsf{P} = \mathsf{NP}$ is one of *the* big open problems in computer science. Most researchers believe that these classes are different, but there is no valid proof so far. The best that we can show is

$$\mathsf{NP} = \bigcup_{i \in \mathbb{N}} \mathsf{NTime}(\mathrm{O}(n^i)) \subseteq \bigcup_{i \in \mathbb{N}} \mathsf{DTime}(2^{\mathrm{O}(n^i)}) =: \mathsf{EXP},$$

that is, nondeterministic polynomially time bounded Turing machines can be simulated by deterministic poly-exponential time bounded ones.

Excursus: The millenium prize problems

The question whether P equals NP is one of the seven millenium prize problems of the Clay mathematics institute. (`www.claymath.org`). If you settle this question, you get \$1000000 (and become famous, at least as famous as a computer scientist can become).
Gerhard Woeginger's P-versus-NP webpage (`www.win.tue.nl/~gwoegi/P-versus-NP.htm`) keeps track of the outgrowth.

22.1 Problems in P

Here is one important problem in P. You may consult your favourite book on algorithms for many other ones.

s-t-CONN is the problem whether a directed graph has a path from a given source node s to a target node t:

$$s\text{-}t\text{-CONN} = \{(G,s,t) \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

(G,s,t) is an encoding of the graph G and the source and target nodes s and t. A reasonable encoding would be the following: All nodes are represented by numbers $1,\ldots,n$, written down in binary. We encode an edge by $[\text{bin}(i),\text{bin}(j)]$. We encode the whole graph by building a large pair that consists of $\text{bin}(n)$, $\text{bin}(s)$, $\text{bin}(t)$, and the encodings of all edges, using our pairing function. Since we only talk about polynomial time computability, the concrete encoding does not matter, and we will not specify the encoding in the following.

We will also just write (G,s,t) or G and will not apply an encoding function. You are now old enough to distinguish whether we mean the graph G itself or its encoding.

Theorem 22.2 s-t-CONN $\in$ P.

22.2 NP and certificates

Beside the definition of NP above, there is an equivalent one based on *verifiers*. We call a Turing machine a *polynomial time Turing machine* if it is p time bounded for some polynomial p.

Definition 22.3 *A deterministic polynomial time Turing machine M is called a* polynomial time verifier *for $L \subseteq \{0,1\}^*$, if there is a polynomial p such that the following holds:*

1. *For all $x \in L$ there is a $c \in \{0,1\}^*$ with $|c| \leq p(|x|)$ such that M accepts $[x,c]$.*

2. *For all $x \notin L$ and all $c \in \{0,1\}^*$, M on input $[x,c]$ reads at most $p(|x|)$ bits of c and always rejects $[x,c]$.*

We denote the language L that M verifies by $V(M)$.

The string c serves as a *certificate* (or *witness* or *proof*) that x is in L. A language L is verifiable in polynomial time if each x in L has a polynomially long proof that $x \in L$. For each x not in L no such proof exists.

Note that the language $V(M)$ that a verifier verifies is not the language that it accepts as a "normal" Turing machine. $L(M)$ can be viewed as a binary relation, the pairs of all (x,c) such that M accepts $[x,c]$.

Theorem 22.4 *$L \in$ NP iff there is a polynomial time verifier for L.*

Proof. We only prove the "$\Longrightarrow$"-direction. Since L is in NP there is a nondeterministic Turing machine M whose time complexity is bounded by some polynomial p such that $L(M) = L$. We may assume w.l.o.g. that in each step, M has at most two nondeterministic choices. We construct a polynomial time verifier V for L. V has one more tape than M. Let $[x,c]$ be the input for V. On the additional tape, V marks $p(|x|)$ cells. (This is possible in polynomial time, since p is time constructible.) Then it copies the first $p(|x|)$ symbols of c into these marked cells. Now V simulates M on x step by step. In each simulated step, it reads one of the bits of c on the additional tape. If M has a nondeterministic choice, V uses the bit of c read in this step to chose one of the two possibilites M has. (To this aim, we order the tuples in the relation δ arbitrarily. If the bit read is 0, then we take the choice which appears before the other one in this ordering. If the bit read is 1, we take the other choice.) In this way, c specifies one path in the computation tree T of M on x. Now if $x \in L$, then there is one path in T of length at most $p(|x|)$ that is accepting. Let c be the bit string that corresponds to this path. Then V accepts $[x,c]$. If $x \notin L$, then no such path exists and hence V will not accept $[x,c]$ for any c. Cleary, the running time of V is bounded by $O(p(|x|))$. ∎

Exercise 22.1 *Prove the other direction of Theorem 22.4*

22.3 Problems in NP

There is an abundance of problems in NP. We here just cover the most basic ones (most likely, even less).

A *clique* of a graph $G = (V,E)$ is a subset C of V such that for all $u,v \in C$ with $u \neq v$, $\{u,v\} \in E$. A clique C is called a *k-clique* if $|C| = k$. Clique is the following language:

$$\text{Clique} = \{(G,k) \mid G \text{ is an undirected graph with a } k\text{-clique}\}.$$

A *vertex cover* of a graph $G = (V, E)$ is a subset C of V such that for each edge $e \in E$, $e \cap C \neq \emptyset$. (Recall that edges of an undirected graph are sets of size two. Thus this condition means that every edge is covered by at least one vertex in C.) VC is the following problem:

$$\text{VC} = \{(G, k) \mid G \text{ is an undirected graph that has a vertex cover of size} \leq k\}.$$

Subset-Sum is the following problem:

$$\text{Subset-Sum} = \{(x_1, \ldots, x_n, b) \mid x_1, \ldots, x_n, b \in \mathbb{N} \text{ and there is an } I \subseteq \{1, \ldots, n\} \text{ with } \sum_{i \in I} x_i = b.\}$$

Let $G = (V, E)$ be a graph and $V = \{v_1, \ldots, v_n\}$. G has a *Hamiltonian cycle* if there is a permutation π such that for all $1 \leq i < n$, $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ and $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$, i.e., there is a cycle that visits each vertex of V exactly once. HC is the following problem:

$$\text{HC} = \{G \mid G \text{ has a Hamiltonian cycle}\}.$$

Next we consider a weighted complete graph $G = (V, \binom{V}{2}, w)$ where $\binom{V}{2}$ denotes all subsets of V of size 2 and $w : \binom{V}{2} \to \mathbb{N}$ assigns to each edge a nonnegative weight. The weight of a Hamiltonian cycle is the weight of the edges contained in it, i.e., $\sum_{i=1}^{n-1} w(\{v_{\pi(i)}, v_{\pi(i+1)}\}) + w(\{v_{\pi(n)}, v_{\pi(1)}\})$. The *traveling salesman problem* is the following problem:

$$\text{TSP} = \{(G, b) \mid G \text{ is a weighted graph with a Hamiltonian cycle of weight} \leq b\}.$$

You can think of a truck that has to deliver goods to different shops and we want to know whether a short tour exists.

Let $x_1, \ldots, x_n$ be Boolean variables, i.e., variables that can take values 0 and 1, interpreted as false and true. A *literal* is either a variable x_i or its negation $\bar{x}_i$. A *clause* is a disjunction of literals $\ell_1 \vee \cdots \vee \ell_k$. k is the length of the clause. A *formula in conjunctive normal form* (formula in CNF for short) is a conjunction of clauses $c_1 \wedge \cdots \wedge c_m$. An *assignment* a is a mapping that assigns each variable a value in $\{0, 1\}$. Such an assignment extends to literals, clauses, and formulas in the obvious way. A formula is called *satisfiable*, if there is an assignment such that the formula attains the value 1. Such an assignment is called a satisfying assignment. If all assignments are satisfying, then the formula is called a *tautology*.

The satisfiability problem, the mother of all problems in NP, is the following problem:

$$\text{SAT} = \{\phi \mid \phi \text{ is a satisfiable formula in CNF}\}.$$

A formula in CNF is in ℓ-CNF, if all its clauses have length at most ℓ. ℓSAT is the following problem:

$$\ell\text{SAT} = \{\phi \mid \phi \text{ is a satisfiable formula in } \ell\text{-CNF}\}.$$

Golden rule of nondeterminism

Nondeterminism is interesting
because it characterizes important problems.

We do not know any physical equivalent to nondeterminism. As far as I know, nobody has been built a nondeterministic Turing machine. But NP is an interesting class because it contains a lot of important problems.

Theorem 22.5 Clique, VC, Subset-Sum, HC, TSP, SAT, ℓSAT $\in$ NP.

Proof. We show that all of the problems have a polynomial time verifier. Let's start with Clique. On input $[x, y]$, a verifier M for Clique first checks whether x is an encoding of the form (G, k). If not, M rejects. It now interprets the string y as a list of k nodes of G, for instance such an encoding could be $\text{bin}(i_1)\$ \ldots \$\,\text{bin}(i_k)$ where $i_1, \ldots, i_k$ are nodes of G. (Since $y \in \{0,1\}^*$, we would then map for instance $0 \mapsto 00$, $1 \mapsto 01$, and $\$ \mapsto 11$.) If y is not of this form, then M rejects. If y has this form, then M checks whether $\{i_j, i_h\}$ is an edge of G, $1 \leq j < h \leq k$. If yes, then M accepts, otherwise it rejects.

We have to show that there is a y such that $[x, y] \in L(M)$ iff $x = (G, k)$ for some graph G that has a k-clique. Assume that $x = (G, k)$ for some graph G that has a k-clique. Then a list of the nodes that form a clique is a proof that makes M accept. On the other hand, if G has no k-clique or x is not a valid encoding, then no proof will make M accept.

For SAT and ℓSAT, an assignment to the variables that satisfies the formula is a possible proof. For VC, a subset of the nodes of size less than k that covers all edges is a possible proof. For Subset-Sum, it is the set of indices I, for HC and TSP, it is the appropriate permutation. The rest of the proof is now an easy exercise. ∎

Decision versus Verification

Cum grano salis:

P is the class of languages L for which we can efficiently decide "$x \in L$?".

NP is the class of languages L for which we can efficiently verify whether a proof for "$x \in L$!" is correct or not.

23 Reduction and completeness

NP contains a lot of interesting problems for which we would like to have efficient algorithms. But most researchers believe that the classes P and NP do not coincide, that is, there is a language $L \in \mathsf{NP}$ such that $L \notin \mathsf{P}$. But we are far from having a proof for this. Instead, we try to identify the "hardest" languages in NP, the so-called NP-complete languages. If we can show that a problem is NP-complete, then this is a strong indication that it does not have a deterministic polynomial time algorithm.

23.1 Polynomial time reductions

Definition 23.1 *Let $L, L' \subseteq \Sigma^*$.*

1. *A function $f : \Sigma^* \to \Sigma^*$ is called a* many-one polynomial time reduction *from L to L' if f is polynomial time computable and*

$$\text{for all } x \in \Sigma^*: \quad x \in L \iff f(x) \in L'.$$

2. *L is* (many-one) polynomial time reducible *to L' if there is a many-one polynomial time reduction from L to L'. We denote this by $L \leq_{\mathrm{P}} L'$.*

Compared to recursive many-one reductions, the function f now shall be polynomial time computable. The reason is that f shall preserve polynomial time computability.

Lemma 23.2 *If $L \leq_{\mathrm{P}} L'$ and $L' \in \mathsf{P}$, then $L \in \mathsf{P}$.*

Proof. Let f be a polynomial time reduction from L to L'. Let M' be a polynomial time deterministic Turing machine with $L' = L(M')$. We construct a polynomial time deterministic Turing machine M for L as follows: On input x, M first computes $f(x)$ and then simulates M' on $f(x)$. M accepts if M' accepts, and rejects otherwise.

First of all, $L(M) = L$, because $x \in L \iff f(x) \in L'$. It remains to show that M is indeed polynomial time bounded. To see this, assume that f is $p(n)$ time computable and M' is $q(n)$ time bounded for polynomials p and q. Since f is $p(n)$ time computable, $|f(x)| \leq p(|x|)$ for all $x \in \Sigma^*$. Thus M is $q(p(n))$ time bounded which is again a polynomial. ∎

Lemma 23.3 *$\leq_{\mathrm{P}}$ is a transitive relation.*

Proof. Let $L \leq_P L'$ and $L' \leq_P L''$. Let f and g be corresponding reductions. We have to show that $L \leq_P L''$. We claim that $g \circ f$ is a polynomial time reduction from L to L''.

First of all, we have for all $x \in \Sigma^*$:

$$x \in L \iff f(x) \in L' \iff g(f(x)) \in L''.$$

It remains to show that $g \circ f$ is polynomial time computable. This is shown as in the proof of Lemma 23.2. ∎

Polynomial time many one reductions versus recursive many one reductions

A many one reduction f from L to L' has the following property:

$$x \in L \iff f(x) \in L' \quad \text{for all } x \in \{0,1\}^*.$$

Recursive many one reduction:

- f is Turing computable.
- f is total.

Polynomial time many one reduction:

- f is polynomial time computable. (This implies that f is total.)

Important properties of recursive many one reducibility:

- $\leq$ is transitive.
- If $L \leq L'$ and $L' \in \mathsf{REC}$ (or RE), then $L \in \mathsf{REC}$ (or RE)

Important properties of polynomial time many one reductions:

- $\leq_P$ is transitive.
- If $L \leq_P L'$ and $L' \in \mathsf{P}$, then $L \in \mathsf{P}$.

23.2 NP-complete problems

Definition 23.4 *1. A language L is* NP-hard *if for all $L' \in \mathsf{NP}$, $L' \leq_P L$.*

2. L is called NP-complete*, if L is* NP*-hard and $L \in \mathsf{NP}$.*

Lemma 23.5 *If L is* NP*-hard and $L \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.*

Proof. Let $L' \in \mathsf{NP}$. Since L is NP-hard, $L' \leq_{\mathrm{P}} L$. By Lemma 23.2, $L' \in \mathsf{P}$. ∎

How can we show that a language is NP-hard? Once we have identified one NP-hard language, the following lemma provides a way to do so.

Lemma 23.6 *If L is NP-hard and $L \leq_{\mathrm{P}} L'$, then L' is NP-hard.*

Proof. Let $L'' \in \mathsf{NP}$. Since L is NP-hard, $L'' \leq_{\mathrm{P}} L$. Since $\leq_{\mathrm{P}}$ is transitive, $L'' \leq_{\mathrm{P}} L'$. Thus L' is NP-hard, too. ∎

It is the famous Cook–Karp–Levin theorem that provides a first NP-complete problem. We defer the proof of it to the next chapters.

Theorem 23.7 (Cook–Karp–Levin) [1] SAT *is NP-complete.*

Excursus: Cook, Karp, or Levin?

In his original paper, Steve Cook did not talk about satisfiability at all, he always talked about tautologies and showed that this problem was NP-complete. This problem is co-NP-complete and it is a big open question whether it is also NP-complete. So how can Steve Cook talk about NP and tautologies? The reason is that he uses a coarser kind of reductions, Turing reductions, instead of many-one reductions. But essentially all his Turing reductions are many-one. This was pointed out by Richard Karp who also showed that many other problems are NP-complete under many-one reductions.

Leonid Levin, a poor guy from the former Soviet Union, invented at the same time as Steve Cook and Richard Karp a similar theory of NP-completeness. Since the cold war was really cold at this time, western scientists became aware of his findings more than a decade later. (He also did not get a Turing award.)

It is rather easy to show that our problem AFSAT is NP-complete.

Lemma 23.8 SAT $\leq_{\mathrm{P}}$ AFSAT.

Proof. Let ϕ be a Boolean formula in CNF with n variables. We construct an arithmetic formula F_ϕ such that every satisfying assignment $a \in \{0,1\}^n$ of ϕ is an assignment such that $a(F) = 1$ and every non-satisfying assignment is an assignment such that $a(F) = 0$. (Note that we interpret 0 and 1 as Boolean values and as integers.)

We construct this formula along the structure of formulas in CNF. Let ℓ be a literal. If $\ell = x$, then $F_\ell = x$. If $\ell = \bar{x}$, then $F_\ell = 1 - x$.

If $c = \ell_1 \vee \cdots \vee \ell_k$ is a clause, then $F_c = 1 - (1 - F_{\ell_1}) \cdots (1 - F_{\ell_k})$. F_c evaluates to 1 iff one of the F_{ℓ_i} evaluates to 1.

[1] This theorem is usually called Cook's theorem. Less conservative authors call it Cook–Levin theorem. The name is Cook–Karp–Levin theorem is my creation, use it at your own risk.

Finally, if $\phi = c_1 \wedge \cdots \wedge c_m$ is a conjunction of clauses, then $F_\phi = F_{c_1} \cdots F_{c_m}$. F_ϕ evaluates to 1 if all F_{c_i} evaluate to 1.

Thus, $\phi \mapsto F_\phi$ is the desired reduction. It is easy to see that this mapping is polynomial time computable. ■

Invalid encodings

In the proof above assumed that the input to the reduction is (an encoding of) a formula in CNF. However, the input can be any string and need not be a valid encoding of a formula. In this case, the input is not in SAT and the reduction needs to map it to a string not in AFSAT. However, this can be easily achieved: We first check whether the input is a valid encoding. This can be done by assumption in polynomial time, since all our encodings are "reasonable". If yes, we apply our reduction form the proof. Otherwise, we map the input to a fixed string not in AFSAT. We will ignore this issue in all further proofs for the ease of presentation.

Let's start to show that the other problems introduced in this chapter are all NP-complete.

Lemma 23.9 *For all $\ell \geq 3$, ℓSAT is NP-complete.* [2]

Proof. We show that SAT $\leq_P$ ℓSAT. It suffices to show this for $\ell = 3$. Let ϕ be a formula in CNF. We have to map ϕ to a formula ψ in 3-CNF such that ϕ is satisfiable iff ψ is satisfiable.

We replace each clause c of length > 3 of ϕ by a bunch of new clauses. Let $c = \ell_1 \vee \cdots \vee \ell_k$ with literals ℓ_κ. Let $y_1, \ldots, y_{k-3}$ be new variables. (We need new variables for each clause.) We replace c by

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\bar{y}_1 \vee \ell_3 \vee y_2) \wedge \cdots \wedge (\bar{y}_{k-4} \vee \ell_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee \ell_{k-1} \vee \ell_k) \quad (23.1)$$

If we do this for every clause of ϕ, we get the formula ψ. This transformation is obviously polynomial time computable.

It remains to show that ϕ is satisfiable iff ψ is satisfiable. Let a be some satisfying assignment for ϕ. We extend this assignment to a satisfying assignment of ψ. Since a satisfies ϕ, for a given clause c, there is one literal, say ℓ_i such that $a(\ell_i) = 1$. If we assign to all y_j with $j < i-1$ the value 1 and to all other y_j the value 0, then all clauses in (23.1) are satisfied. Thus we found a satisfying assignment for ψ. On the other hand, if ψ is satisfiable,

[2] Our proof of the Cook-Karp-Levin theorem will actually show that 3SAT is NP-complete. It is nevertheless very instructive to see the reduction from SAT to 3SAT.

then any satisfying assignment b that satisfies all the clauses in (23.1) has to set one ℓ_i to 1, since the y_j's can only satisfy at most $k-3$ clauses but (23.1) contains $k-2$ clauses. Thus the restriction of b to the variables of ϕ satisfies c and henceforth ϕ. ∎

Exercise 23.1 *Astonishingly,* 2SAT $\in$ P.

1. *Given a formula ϕ in 2-CNF over variables $x_1, \ldots, x_n$, we construct a directed graph $G = (V, E)$ as follows: $V = \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$ is the set of all literals and for each clause $\ell_1 \vee \ell_2$ we add the two edges $(\bar{\ell}_1, \ell_2)$ and $(\bar{\ell}_2, \ell_1)$ to E. Show the following: ϕ is satisfiable iff for all $1 \leq \nu \leq n$, there is no directed cycle that contains x_ν and $\bar{x}_\nu$.*
2. *Conclude that* 2SAT $\in$ P.

Reducing SAT to ℓSAT was not too hard. (Or at least, it does not look too unreasonable that one can find such a reduction.) Reducing SAT or ℓSAT to Clique, for instance, looks much harder, since these problems seem to be completely unrelated. First such reductions look like art, but nowadays it has become routine work (with some exceptions) and there is a huge toolbox available.

Lemma 23.10 ℓSAT $\leq_{\mathrm{P}}$ Clique.

Proof. Let ϕ be a formula in 3-CNF. We may assume that each clause of ϕ has exactly three literals by possibly repeating some literals. Let

$$\phi = (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge \cdots \wedge (\ell_{m,1} \vee \ell_{m,2} \vee \ell_{m,3})$$

We have to construct a pair (G, k) such that $G = (V, E)$ has a k-clique iff ϕ is satisfiable. We set $V = \{(1,1), (1,2), (1,3), \ldots, (m,1), (m,2), (m,3)\}$, one node for each literal of a clause. E is the set of all $\{(i,s), (j,t)\}$ such that $i \neq j$ and $\ell_{i,s} \neq \bar{\ell}_{j,t}$. In other words, there is *no* edge $\{(i,s), (j,t)\}$ iff $\ell_{i,s}$ and $\ell_{j,t}$ cannot be simultaneously set to 1 (because one is the negation of the other). Finally, we set $k = m$.

If ϕ is satisfiable, then there is a satisfying assignment for ϕ, i.e., an assignment that assigns to at least one literal of each clause the value 1. Let $\ell_{1,s_1}, \ldots, \ell_{m,s_m}$ be these literals. Then $(1, s_1), \ldots, (m, s_m)$ form a clique of size m in G.

Conversely, if G has a clique of size k, then it is of the form $(1, s_1), \ldots, (m, s_m)$, because there is no edge between (i, s) and (i, t) for $s \neq t$. Then we can set all the literals $\ell_{1,s_1}, \ldots, \ell_{m,s_m}$ to 1 and hence ϕ is satisfiable.

The mapping $\phi \mapsto (G, k)$, is obviously polynomial time computable. ∎

Lemma 23.11 Clique $\leq_{\mathrm{P}}$ VC.

Proof. For a graph $G = (V, E)$, let $\bar{G} = (V, \binom{V}{2} \setminus E)$ be its complement, i.e, e is an edge of G iff e is not an edge of $\bar{G}$.

Let C be a clique of G. We will show below that $V \setminus C$ is an vertex cover of $\bar{G}$. Conversely, if D is a vertex cover of $\bar{G}$, then $V \setminus D$ is a clique of G.

In particular, G has a clique of size at least k iff $\bar{G}$ has an vertex cover of size at most $n - k$. Thus $(G, k) \mapsto (\bar{G}, n - k)$ is the desired reduction. This reduction is of course polynomial time computable.

If C is a clique of G, then there are no edges between nodes of C in $\bar{G}$. Thus the nodes of $V \setminus C$ cover all edges of $\bar{G}$, since every edge in $\bar{G}$ has at least one node not in C. Conversely, if D is a vertex cover of $\bar{G}$, then there are no edges between then nodes of $V \setminus D$, because otherwise D would not be a vertex cover. Thus $V \setminus D$ is a clique in G. ∎

Lemma 23.12 3SAT $\leq_P$ Subset-Sum.

The proof of this lemma is more complicated; we defer it to the next chapter.

Exercise 23.2 *Consider the following dynamic programming approach to* Subset-Sum. *Let $x_1, \ldots, x_n, b$ be the given instance. We define a predicate $P(\nu, \beta)$ for $1 \leq \nu \leq n$ and $0 \leq \beta \leq b$ by*

$$P(\nu, \beta) = \begin{cases} 1 & \text{if there is an } I \subseteq \{1, \ldots, \nu\} \text{ with } \sum_{i \in I} x_i = \beta \\ 0 & \text{otherwise} \end{cases}$$

1. *Show that $P(\nu, \beta) = P(\nu - 1, \beta) \vee P(\nu - 1, \beta - x_\nu)$.*
2. *Design an algorithm with running time $O(nb)$ for* Subset-Sum.
3. *Does this show that $\mathsf{P} = \mathsf{NP}$? (See also Chapter 27.)*

Lemma 23.13 3SAT $\leq_P$ HC.

This proof is again deferred to the next section.

Lemma 23.14 HC $\leq_P$ TSP.

Proof. Let $G = (V, E)$ be an input of HC. An input of TSP is a weighted complete graph $H = (V, \binom{V}{2}, w)$. We assign to edges from G weight 1 and to "nonedges" weight 2, i.e,

$$w(e) = \begin{cases} 1 & \text{if } e \in E \\ 2 & \text{if } e \notin E \end{cases}$$

By construction, G has a Hamiltonian cycle iff H has a Hamiltonian cycle of weight n. ∎

Theorem 23.15 Clique, VC, Subset-Sum, HC, TSP, SAT, 3SAT, *and* AFSAT *are* NP*-complete.*

The proof of the theorem follows from the lemmas in this chapter.

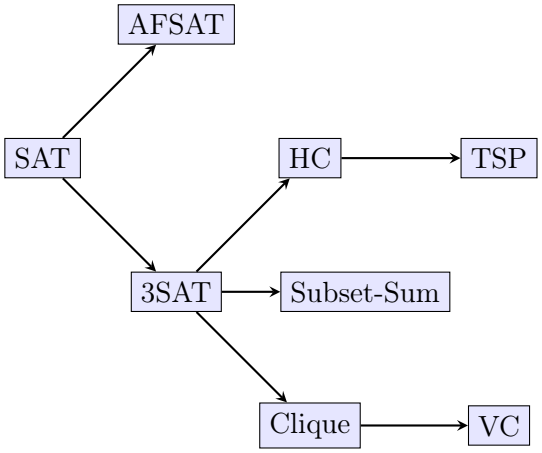


Figure 23.1: The reduction scheme. An arrow from A to B means that we proved $A \leq_{\mathrm{P}} B$.

24 More reductions

In this chapter, we construct the two missing reductions from the last chapter. They are more complicated than the ones in the last chapter, but now you should be old enough to understand them. When you see such reductions for the first time, they look like complicated magic, but constructing them has become a routine job, with some notable exceptions.

24.1 Subset-Sum

We first start with the proof of Lemma 23.12. As Exercise 23.2 suggests, the instances created by the reduction will use large numbers, that is, numbers whose size is exponential in the number of clauses of the Boolean formula (or equivalently, the length of the binary representation will we polynomial in m).

Proof of Lemma 23.12. Let ϕ be a formula in 3-CNF. We have to construct an instance of Subset-Sum, i.e., numbers $a_1, \ldots, a_t, b$ such that there is a subset $I \subseteq \{1, \ldots, t\}$ with $\sum_{i \in I} a_i = b$ iff ϕ is satisfiable.

Let $x_1, \ldots, x_n$ be the variables of ϕ. Let $c_1, \ldots, c_m$ be the clauses of ϕ. For each literal ℓ we will construct a number $a(\ell)$ as follows: The number $a(\ell)$ is of the form $a_0(\ell) + 10^n \cdot a_1(\ell)$. The first part $a_0(\ell)$ is the variable part, the second part $a_1(\ell)$ is the clause part. For a variable x_ν, let $c_{\mu_1}, \ldots, c_{\mu_{s_\nu}}$ be the clauses in which it appears positively, or in other words, $c_{\mu_1}, \ldots, c_{\mu_{s_\nu}}$ are the clauses that contain the literal x_ν. Then

$$a(x_\nu) = 10^{\nu-1} + 10^n(10^{\mu_1 - 1} + \cdots + 10^{\mu_{s_\nu} - 1}).$$

For a literal $\bar{x}_\nu$, let $c_{\bar{\mu}_1}, \ldots, c_{\bar{\mu}_{\bar{s}_\nu}}$ are the clauses that contain the literal $\bar{x}_\nu$. Then

$$a(\bar{x}_\nu) = 10^{\nu-1} + 10^n(10^{\bar{\mu}_1 - 1} + \cdots + 10^{\bar{\mu}_{\bar{s}_\nu} - 1}).$$

Choosing $a(x_\nu)$ indicates that we set x_ν to 1. Choosing $a(\bar{x}_\nu)$ indicates that we set $\bar{x}_\nu$ to 1, i.e., x_ν to 0. Of course we can set x_ν either to 1 or to 0. This means that we shall only be able to select one of $a(x_\nu)$ and $a(\bar{x}_\nu)$. Thus in the "target number" $b = b_0 + 10^n b_1$, be set $b_0 = 1 + 10 + \cdots + 10^{n-1}$.

The numbers $a(x_\nu)$ and $a(\bar{x}_\nu)$ have digits 0 or 1. For each position 10^i, there are at most 3 numbers that have digit 1 at position 10^i. In the variable part, this is clear, since only $a(x_i)$ and $a(\bar{x}_i)$ have a 1 in position 10^i. In the clause part, this is due to the fact that each clause consists of at most three literals. Since our base 10 is larger than 3, in the sum of any subset

of $a(x_\nu), a(\bar{x}_\nu)$, $1 \leq \nu \leq n$, no carry can occur. (We could have chosen a smaller base but 10 is so convenient.) This means that any sum that yields b_0 in the lower n digits either contains $a(x_\nu)$ or $a(\bar{x}_\nu)$, $1 \leq \nu \leq n$. This ensures consistency, that means, we can read off a corresponding assignment from the chosen numbers.

Finally, we have to ensure that the assignment is also satisfying. This is done by choosing b_1 properly. Each clause should be satisfied, so a first try would be to set $b_1 = 1 + 10 + \cdots + 10^{m-1}$. But a clause c_μ could be satisfied by two or three literals, in this case the digit of $10^{n-1+\mu}$ is 2 or 3. The problem is that we do not know in advance whether it is 1, 2, or 3. Therefore, we set $b_1 = 3(1+10+\cdots+10^{m-1})$ and introduce "filler numbers" $c_{\mu,1} = c_{\mu,2} = 10^{n-1+\mu}$, $1 \leq \mu \leq m$. We can use these filler numbers to reach the digit 3 in position $10^{n-1+\mu}$. But to reach 3, at least one 1 has to come from an $a(x_\nu)$; thus the clause is satisfied if we reach 3.

Overall, the considerations above show that ϕ has a satisfying assignment iff a subset of $a(x_\nu), a(\bar{x}_\nu)$, $1 \leq \nu \leq n$, and $c_{\mu,1}, c_{\mu,2}$, $1 \leq \mu \leq m$, sums up to b. Thus the reduction above is a polynomial time many one reduction from 3SAT to Subset-Sum. ∎

24.2 Hamiltonian Cycle

In order to prove Lemma 23.13, we introduce an intermediate problem, *directed Hamiltonian cycle*. Here we consider directed graphs $G = (V, E)$. The edges are now ordered pairs, i.e., elements of $V \times V$. We say that (u, v) is an edge from u to v. Let $V = \{v_1, \ldots, v_n\}$. Now a (directed) Hamiltonian cycle is a permutation such that $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ for all $1 \leq i < n$ and $(v_{\pi(n)}, v_{\pi(1)}) \in E$. That is, it is a cycle that visist each node exactly once and all the edges in the cycle have to point in the same direction.

Dir-HC is a generalization of HC. To each undirected graph H corresponds a directed one, G, in a natural way: Each undirected edge $\{u, v\}$ is replaced by two directed ones, (u, v) and (v, u). Any Hamiltonian cycle in G induces a Hamiltonian cycle in H in a natural way. Given H, G can be computed easily. Thus HC $\leq_P$ Dir-HC. But we can also show the converse.

Lemma 24.1 Dir-HC $\leq_P$ HC.

Proof. Let $G = (V, E)$ be a directed graph. We construct a undirected graph $G' = (V', E')$ such that G has a Hamiltonian cycle iff G' has a Hamiltonian cycle.

G' is obtained from G as follows: For every node $v \in V$, we introduce three nodes v_{in}, v, v_{out} and connect v_{in} with v and v_{out} with v.[1] Then for

[1] Such a thing is usually called a *gadget*. We replace a node or edge (or something like this) by a small graph (or something like this). The term gadget is used in an informal way, there is no formal definition of a gadget.

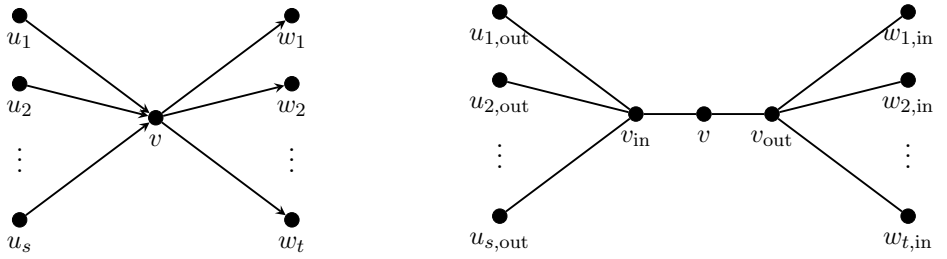


Figure 24.1: The gadget for the reduction: $u_1, \ldots, u_s$ are the nodes such that there is an edge $(u_i, v) \in E$, that is, an edge entering v. The nodes $w_1, \ldots, w_t$ are the nodes such that there is an edge $(v, w_j) \in E$, that is, an edge leaving v. The two lists $u_1, \ldots, u_s$ and $v_1, \ldots, v_t$ need not be disjoint. The righthand side show the gadget. Every node is replaced by three nodes, v, v_{in}, v_{out}. For every directed edge (x, y), we add the undirected edge $\{x_{\text{out}}, y_{\text{in}}\}$.

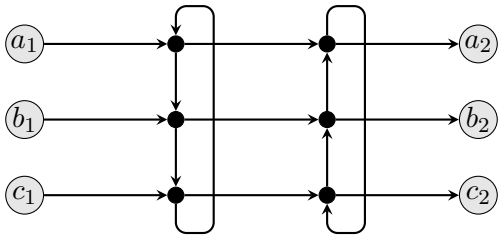


Figure 24.2: The gadget for the reduction of 3SAT to Dir-HC.

every directed edge $(x, y) \in E$, we add the undirected edge $\{x_{\text{out}}, y_{\text{in}}\}$ to E'. Figure 24.1 shows this construction.

Given G, we can construct G' in polynomial time. Thus what remains to show is the follow: G has a Hamiltonian cycle iff G' has a Hamiltonian cycle. Assume that G has a Hamiltonian cycle C. Then we get a Hamiltonian cycle of G' as follows. For every edge (x, y) of C, we take the edge $\{x_{\text{out}}, y_{\text{in}}\}$. Furthermore We add the $\{v_{\text{in}}, v\}$ and $\{v, v_{\text{out}}\}$. This gives a Hamiltonian cycle of G'.

For the converse direction, observe that every node in a Hamiltonian cycle is incident with two edges. Since every node v is only incident with two edges in G', the edges $\{v_{\text{in}}, v\}$ and $\{v, v_{\text{out}}\}$ have to be in a Hamiltonian cycle in G'. The other edges of the Hamiltonian cycle in G' induce a Hamiltonian cycle in G. ■

For 3SAT, we need a more complicated gadget.

Lemma 24.2 *Let G be the graph in Figure 24.2.*

1. *For every nonempty subset $S \subseteq \{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$, there are node disjoint path from s to t for all $(s, t) \in S$ such that all inner nodes of G lie on one of these path.*

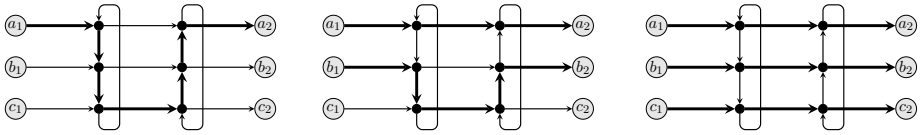


Figure 24.3: The paths connecting the nodes in S.

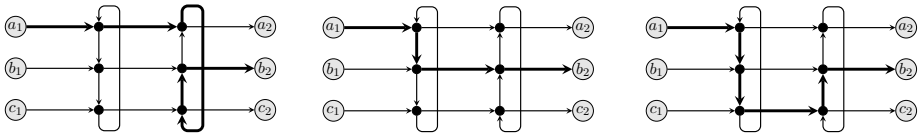


Figure 24.4: No matter how we connect a_1 with b_2, there are always inner nodes left that are not covered and cannot be covered by other paths.

2. For any other subset $T \subseteq \{a_1, b_1, c_1\} \times \{a_2, b_2, c_2\}$, no such paths exist.

Proof. We start with the first part: Figure 24.3 shows these paths in the case of one, two, or three pairs. Only the number of pairs in S matters, since the structure of the gadget G is invariant under simultaneous cyclic shifts of the nodes a_1, b_1, c_1 and a_2, b_2, c_2.

For the second part consider any other pair. Since the gadget is invariant under cyclic shifts, it is enough to consider the pair (a_1, b_2) and the pair (a_2, b_1). Figure 24.4 shows all possibilities how to connect a_1 with b_2. In each case, inner nodes are not covered and it is not possible to cover all of them with other paths. The other pair is an exercise. ■

Exercise 24.1 *Draw the corresponding figures for the pair (a_2, b_1).*

Lemma 24.3 3SAT $\leq_{\mathrm{P}}$ Dir-HC.

Proof. For this proof, we have to do the following: Given a formula ϕ in 3-CNF, we have to map it to a directed graph G (depending on ϕ) such that ϕ is satisfiable iff G has a Hamiltonian cycle. Every variable of ϕ will be represented by a node, every clause will be represented by a gadget from Figure 24.2.

Let $x_1, \ldots, x_n$ be the variables of ϕ and $c_1, \ldots, c_m$ be its clauses. We call the nodes representing the variables $x_1, \ldots, x_n$, too. There will be two paths from x_i to x_{i+1} for each $1 \leq i < n$ and two paths from x_n to x_1. One corresponds to the fact that x_i is set to 1, the other one corresponds to the case that x_i is set to 0. Let C_j be the gadget that represents c_j. Assume that x_i occurs positively in clauses $c_{j_1}, \ldots, c_{j_s}$ and negatively in the clauses $c_{k_1}, \ldots, c_{k_t}$. Then an edge goes from x_i to C_{j_1}. If it is the first literal in c_{j_1}, then this edge goes to the node a_1, if it is the second, then it enters through

b_1, and if it is the third, it uses c_1. Then there is an edge from C_{j_1} to C_{j_2}. It leaves C_{j_1} through the node corresponding to the entry node. I.e., if we entered C_{j_1} through a_1, we also leave to through a_2, and so on. Finally, the edge leaving C_{j_s} goes to x_{i+1}. The second paths is constructed in the same manner and goes through $C_{k_1}, \ldots, C_{k_t}$. Every clause gadget appears on one, two, or three path, depending on the number of its literals. Finally, we remove all the a_i, b_i, and c_i nodes, $i = 1, 2$. For each such node, if there is one edge going into it and a second one leaving it, then we replace these two edges by one edge going from the start node of the first edge to the end node of the second edge. When such a node is only incident with one edge, we remove it and its edge completely.

Figure 24.5 shows an example. x_2 is the first literal of c_3, the third of c_5, and the first of c_8. $\bar{x}_1$ is the second literal of c_2.

Let G be the graph constructed from ϕ. G can certainly constructed in polynomial time. So it remains to show that ϕ has a satisfying assignment iff G has a Hamiltonian cycle.

For the "$\Rightarrow$"-direction, let a be a satisfying assignment of ϕ. We construct a Hamiltonian cycle as follows. If $a(x_i) = 1$, we use all the edges of the paths to x_{i+1} that contain the clauses in which x_i occurs positively. In the other case, we use the other path. Since a is a satisfying assignment, at least one of the inner node of C_i that were right of a_1, b_1, or c_1 is incident with one of these edges. And by construction, also the corresponding inner nodes that were left of a_2, b_2, or c_2 are. By the first part of Lemma 24.2, we can connect the corresponding pairs such that all inner nodes of the gadget lie on a path. This gives a Hamiltonian cycle of G.

For the converse direction, let H be a Hamiltonian cycle of G. By the second part of Lemma 24.2, when the cycle enters a clause gadget through the inner node that was right to a_1, it leaves it through the inner node left to a_2 and so forth. This means that the next variable node that the cycle visits after x_i is x_{i+1}. Since only one edge can leave x_i, the cycle either goes through the path with positive occurences of x_i or through the path with negative occurences of x_i. In the first case, we set x_i to 1, in the second to 0. Since H is a Hamiltonian cycle, it goes through each clause gadget at least once. Hence this assignment will be a satisfying assignment. ∎

Corollary 24.4 3SAT $\leq_{\mathrm{P}}$ HC.

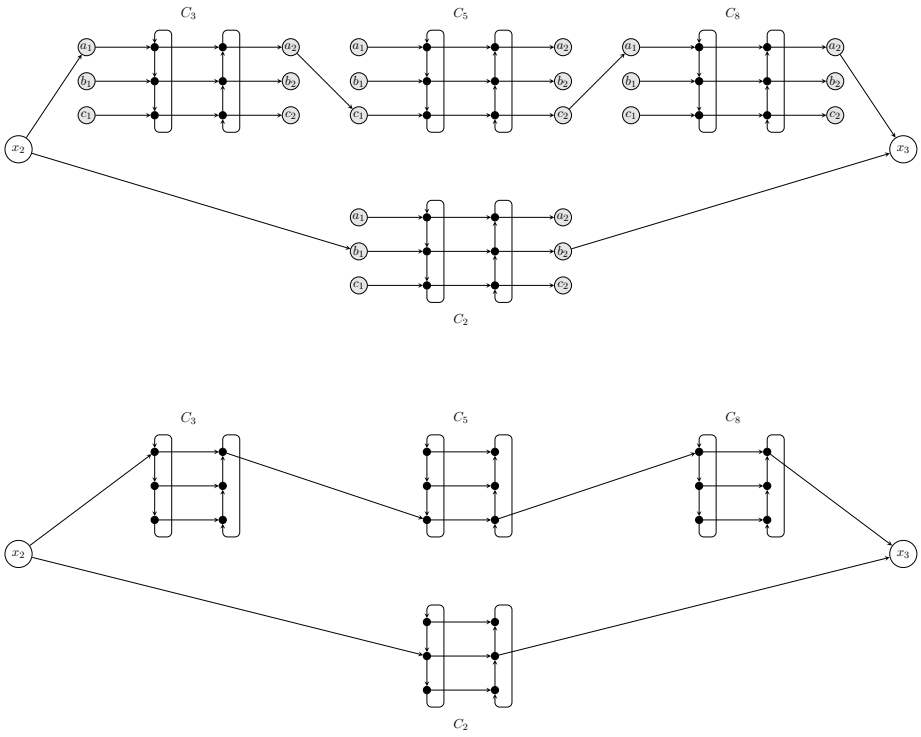
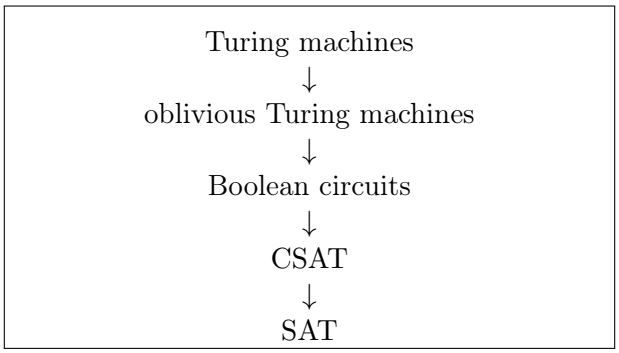


Figure 24.5: An example. The variable x_2 appear positively in the clauses c_3, c_5, and c_8 in the first, third, and first position. It appears negatively in c_2 in the second position. The other a_i, b_i, and c_i nodes of the gadges lie on paths between other variables. Next, all the a_i, b_i, and c_i nodes, $i = 1, 2$ are removed and the two edges incident to them are replaced by one.

V Proof of the Cook–Karp–Levin theorem

Reducing 3SAT to Subset-Sum, for instance, was a hard job, because the problems look totally different. To show that SAT is NP-hard, we have to reduce any language in NP to SAT. The only thing that we know about such an L is that there is a polynomially time bounded nondeterministic Turing machine M with $L(M) = M$. Thus we have to reduce the question whether a Turing machine M accepts a word x to the question whether some formula in CNF is satisfiable. (This makes 3SAT $\leq_{\mathrm{P}}$ Subset-Sum look like a picnic.) The general reduction scheme looks as follows:



As an intermediate concept, we introduce *Boolean circuits*. We show that Boolean circuits can simulate Turing machines. To do so, we have to first make the Turing machine *oblivious*, that means, that on all inputs of a specific length, the Turing machine moves its heads in the same way. Once we know that Boolean circuits can simulate Turing machine, it is rather easy to show that the *circuit satisfiability problem* CSAT (given a circuit C, does it have a satisfying assignment?) is NP-hard. Lastly, we show that CSAT $\leq_{\mathrm{P}}$ SAT.

V.1 Boolean functions and circuits

We interpret the value 0 as Boolean false and 1 as Boolean true. A function $\{0,1\}^n \to \{0,1\}^m$ is called a Boolean function. n is its arity, also called the input size, and m is its output size.

A *Boolean circuit* C with n inputs and m outputs is an acyclic digraph with $\geq n$ nodes of indegree zero and m nodes of outdegree zero. Each node has either indegree zero, one or two. If its indegree is zero, then it is labeled

with $x_1, \ldots, x_n$ or 0 or 1. Such a node is called an input node. If a node has indegree one, then it is labeled with $\neg$. Such a node computes the Boolean negation. If a node has indegree two, it is labeled with $\vee$ or $\wedge$ and the node computes the Boolean or or Boolean and, respectively. The nodes with outdegree zero are ordered so that we can speak about the first output bit, the second output bit etc. The nodes in a Boolean circuit are sometimes called gates, the edges are called wires.

The *depth* of a node v of C is the length of a longest path from a node of indegree zero to v. (The length of a path is the number of edges in it.) The depth of v is denoted by $\text{depth}(v)$. The depth of C is defined as $\text{depth}(C) = \max\{\text{depth}(v) \mid v \text{ is a node of } C\}$. The size of C is the number of nodes in it and is denoted by $\text{size}(C)$.

Such a Boolean circuit C computes a Boolean function $\{0,1\}^n \to \{0,1\}^m$ as follows. Let $\xi \in \{0,1\}^n$ be a given input. With each node, we associate a value $\text{val}(v,\xi) \in \{0,1\}$ computed at it. If v is an input node, then $\text{val}(v,\xi) = \xi_i$, if v is labeled with x_i. If v is labeled with 0 or 1, then $\text{val}(v,\xi)$ is 0 or 1, respectively. This defines the values for all nodes of depth 0. Assume that the value of all nodes of depth d are known. Then we compute $\text{val}(v,\xi)$ of a node v of depth $d+1$ as follows: If v is labeled with $\neg$ and u is the predecessor of v, then $\text{val}(v,\xi) = \neg\,\text{val}(u,\xi)$. If v is labeled with $\vee$ or $\wedge$ and u_1, u_2 are the predecessors of v, then $\text{val}(v,\xi) = \text{val}(u_1,\xi) \vee \text{val}(u_2,\xi)$ or $\text{val}(v,\xi) = \text{val}(u_1,\xi) \wedge \text{val}(u_2,\xi)$. For each node v, this defines a function $\{0,1\}^n \to \{0,1\}$ computed at v by $\xi \mapsto \text{val}(v,\xi)$. Let $g_1, \ldots, g_m$ be the functions computed at the output nodes (in this order). Then C computes a function $\{0,1\}^n \to \{0,1\}^m$ defined by $\xi \mapsto g_1(\xi)g_2(\xi)\ldots g_m(\xi)$. We denote this function by $C(\xi)$.

The labels are taken from $\{\neg, \vee, \wedge\}$. This set is also called *standard basis*. This standard is known to be complete, that is, for any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there is Boolean circuit (over the standard basis) that computes it. For instance, the CNF of a function directly defines a circuit for it. (Note that we can simulate one Boolean and or or of arity n by $n-1$ Boolean and or or of arity 2.)

Boolean circuits can be viewed as a model of parallel computation, since a node can compute its value as soon as it knows the value of its predecessor. Thus, the depth of a circuits can be seen as the time taken by the circuit to compute the result. Its size measures the "hardware" needed to built the circuit.

Exercise V.1 *Every Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a Boolean circuit of size $2^{O(n)}$. (Remark: This can be sharpened to $(1+\epsilon) \cdot 2^n/n$ for any $\epsilon > 0$. The latter bound is tight: For any $\epsilon > 0$ and any large enough n, there is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ such that every circuit computing f has size $(1-\epsilon)2^n/n$.)*

V.2 Uniform families of circuits

There is a fundamental difference between circuits and Turing machines. Turing machines compute functions with variable input length, e.g., $\{0,1\}^* \to \{0,1\}^*$. Boolean circuits only compute a function of fixed size $\{0,1\}^n \to \{0,1\}^m$. To overcome the problem that circuits compute functions of fixed length, we will introduce families of circuits.

In the following, we will only look at Boolean circuits with one output node, i.e., circuits that decide languages. Most of the concepts and results presented in the remainder of this chapter also work for circuits with more output nodes, that is, circuits that compute functions.

Definition V.1 *1. A sequence $C = (C_n)$ of Boolean circuits such that C_i has i inputs is called a family of Boolean circuits.*

2. C is s size bounded and d depth bounded if $\text{size}(C_i) \leq s(i)$ *and* $\text{depth}(C_i) \leq d(i)$ *for all i.*

3. C computes the function $\{0,1\}^* \to \{0,1\}$ given by $x \mapsto C_{|x|}(x)$. Since we can interpret this as a characteristic function, we also say that C decides a language. We write $L(C)$ for this language.

Families of Boolean circuits can decide nonrecursive languages, in fact any $L \subseteq \{0,1\}^*$ is decided by a familiy of Boolean circuits. To exclude such phenomena, we put some restrictions on the families.

Definition V.2 *1. A family of circuits is called s space and t time constructible, if there is an s space bounded and t time bounded deterministic Turing machine that given input 1^n writes down an encoding of C_n.*

2. A family of circuits C is called polynomial time uniform *if is is constructible in time polynomial in n.*

Polynomial time uniform families of circuits always have polynomially bounded size.

V.3 Simulating Turing machines by families of circuits

If we want to simulate Turing machines by circuits, there is a problem. For two different inputs of the same length n, a Turing machine can do completely different things, on the one input, it could move to the left, on the other it could move to the right. But the same two inputs are fed into the circuit C_n which is static and essentially does the same on all inputs. How can such a

poor circuit simulate the behaviour of the Turing machine on *all* inputs of length n. The idea is to tame the Turing machine.

Definition V.3 *A Turing machine is called* oblivious *if the movement of the heads are the same for all inputs of length n. (In particular, it performs the same number of steps on each input of length n.)*

Lemma V.4 *Let t be time constructible. For every t time bounded deterministic Turing machine M, there is an oblivious $O(t^2)$ time bounded 1-tape deterministic Turing machine S with $L(M) = L(S)$.*

Proof. First, we replace the Turing machine M by a Turing machine that uses a one-sided infinite tape. S is basically the construction from Lemma 20.2. Since t is time constructible, it is also space constructible. On input x, S first marks $t(|x|)$ cells and then simulates M as described in the proof of Lemma 20.2.

To simulate one step of M, it makes a sweep over all the marked $t(|x|)$ cells and not just those visisted by M so far. Furthermore, S halts after exactly simulating $t(|x|)$ steps of M. If M halted before, then S just performs some dummy steps that do not change anything. In this way, S becomes oblivious. ∎

Lemma V.5 *Let M be a polynomial time bounded oblivious 1-tape deterministic Turing machine with input alphabet $\{0,1\}$. Then there a polynomial time uniform family of circuits C with $L(M) = L(C)$.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$. Let M be t time bounded. We can assume that the set of states of M is a subset of $\{0,1\}^d$ for some constant d. $0 \dots 0$ shall be the start state. Since a circuit can only deal with values 0 and 1, we will represent the tape alphabet by words from $\{0,1\}^c$ for some constant c. 0 will be mapped to $0 \dots 0$ and 1 will be mapped to $1 \dots 1$. (This choice is fairly arbitrary and does not really matter.)

We first built a circuit D which gets an input from $\{0,1\}^d \times \{0,1\}^c$, a state of M and an encoding of a symbol from the tape alphabet, and produces as output again an element from $\{0,1\}^d \times \{0,1\}^c$. If we feed (the encoding of) (q, γ) in to D, we get the output (q', γ') where $\delta(q, \gamma) = (q', \gamma', r)$ for some $r \in \{L, S, R\}$, that is, D computes the transition function (except the direction).

The circuit C_n now works in layers. Each layer consists of d edges that carry the state of M and of $t(n)$ "packets" of c edges, one packet for each cell that M potentially can visit. Between the ith and the $(i+1)$th layer we place a copy of D; the d edges that correspond to the state and the c edges that correspond to the cell from which M reads a symbol in step i. Since M is oblivious, this cell is independent of the input. Into the first layer, we

feed the input bits into these packets of edges that correspond to the n cells that contain the input. In all other packets, we feed constants that encode the blank, say $0\ldots01$. In the edges that correspond to the state we feed the constants that that encode that start state, that is, $0\ldots0$. After the last layer, we feed the edges that carry the state into a small circuit E that outputs 1, iff the encoded state is accepting and 0 otherwise. See Figure V.1 for a sketch of C_n.

On input 1^n, a Turing machine N can construct C_n as follows: C_n has a very regular structure, so N constructs it layer by layer. The circuit D can be "hard-wired"[1] into N, since the size of D is finite. The only problem is to find out where to place D. But since M is oblivious, it suffices to simulate M on 1^n. This simulation also gives us the number of layers that C_n has, namely the number of steps that M performs.

Since M is polynomial time bounded, the family (C_n) is polynomial time uniform. ■

V.4 The proof

Before we show that SAT is NP-hard, we show that a more general problem is NP-hard, the circuit satisfiability problem CSAT which is the following problem: Given (an encoding of) a Boolean circuit C, decide whether there is a Boolean vector ξ with $C(\xi) = 1$.

Theorem V.6 CSAT *is* NP*-hard*

Proof. Let $L \in \mathsf{NP}$ and let M be a polynomial time verifier for it. We can assume that M is oblivious. Let p be the polynomial that bounds the length of the certificates. We can also assume that all certificates y such that M accepts $[x, y]$ have length *exactly* $p(|x|)$. To do so, we can for instance replace each 0 of y by 00 and each 1 by 11 and pad the certificate by appending 01. This doubles the length of the certificates, which is fine.

We saw in Lemma V.5, that for any oblivious polynomial time bounded Turing machine, there is a polynomial time uniform family of polynomial size circuits C_i that decides the same language.

Now our reduction works as follows: Since for each x of length n, all interesting certificates (certificates such that M might accept $[x, y]$) have the same length, all interesting pairs $[x, y]$ have the same length $\ell(n)$, which depends only on n. Given x, we construct $C_{\ell(|x|)}$. Then we construct a circuit with $n + p(n)$ inputs that given x and y computes $[x, y]$ and use its output as the input to $C_{\ell(|x|)}$ Finally, we specialize the inputs belonging to the symbols of the first part of the input to x. Our reduction simply maps x to this circuit.

[1] This means that there is a "subroutine" in N that prints D on the tape

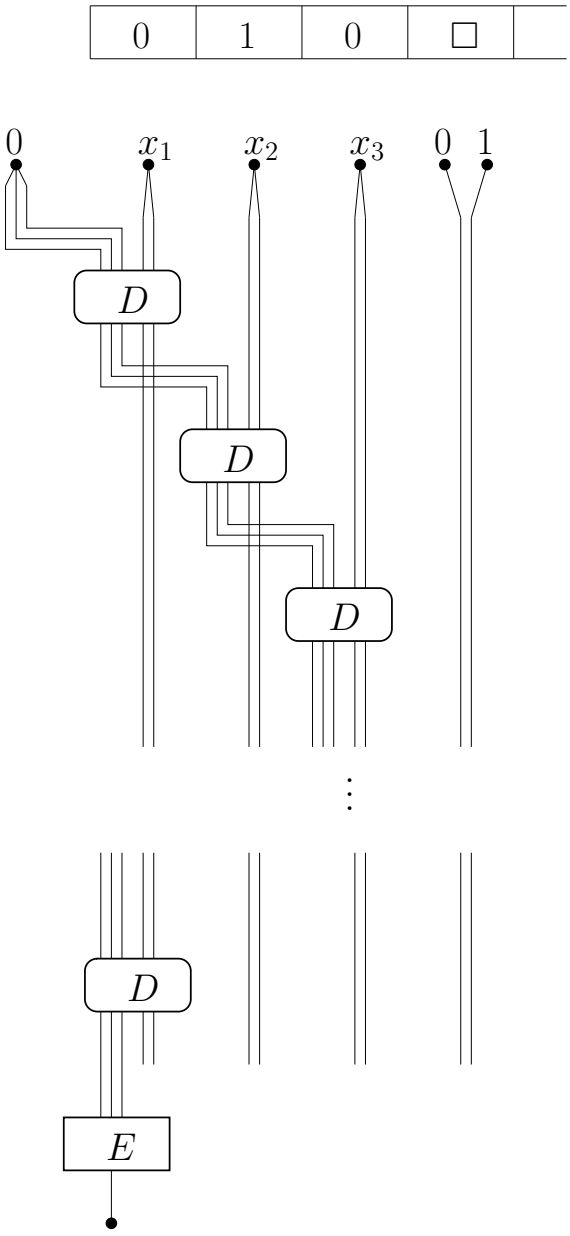


Figure V.1: The circuit C_n that simulates the Turing machine on inputs of length n. At the top, the tape of the Turing machine is shown. The input is 010. The Turing machine moves its head two times to the right during its first two steps. The states are a subset of $\{0,1\}^3$ and the symbols of the tape alphabet Γ are encoded as words from $\{0,1\}^2$. Since $0 \in \Gamma$ is encoded by 00 and $1 \in \Gamma$ is encoded by 11, we just can duplicate the input node x_ν and feed the two edges into D. A blank is represented by 01. There are "edges" at the bottom that do not end in any nodes. They actually do not appear in C_n, we have just drawn them to depict the regular structure of the layers.

By construction, this new circuit C' has an input y with $C'(y) = 1$ if there is a y such that $M([x, y]) = 1$. Thus the mapping $x \mapsto C'$ is a many one reduction of L to CSAT. It is also polynomial time, since C can be constructed in polynomial time. ■

Theorem V.7 SAT *is* NP*-hard.*

Proof. It is sufficient to show CSAT $\leq$ SAT. Let C be the given input circuit and let s be its size. Note that we cannot just expand a circuit into an equivalent formula, since this may result in an expontential blow up.

Instead, we introduce new input variables $a_1, \ldots, a_s$. The idea is that a_j is the output of the jth gate of C. We will construct a formula by constructing a formula in CNF for each gate. The final formula will be the conjunction of these small formulas.

Let gate j be an input gate. We can assume that each variable in C is assigned to exactly one input gate. If the gate is labeled with a variable, we do nothing. a_j models this variable. If j is labeled with a constant, then we add the clause a_j or $\neg a_j$. This forces a_j to be 1 or 0, respectively, in a satisfying assignment.

If gate j is the negation of gate i, our formula contains the expression

$$(a_i \vee a_j) \wedge (\neg a_i \vee \neg a_j).$$

These formula is satisfied iff a_i and a_j get different values.

If gate j is the conjunction of gates h and i, then we add the expression

$$(\neg a_j \vee a_i) \wedge (\neg a_j \vee a_h) \wedge (a_j \vee \neg a_i \vee \neg a_h). \tag{V.1}$$

In a satisfying assignment, either a_j is 0 and at least one of a_i and a_h is 0 or all three are 1.

There is a similar expression for Boolean or.

Finally, we have the additional expression a_s, where s is the output gate. This forces the output to be 1.

Now if there is an input such that $C(x) = 1$, then by construction, we get a satisfying assignment by giving the variables $a_1, \ldots, a_s$ the values computed at the corresponding gates. Conversely, any satisfying assignment also gives an assigment to the input variables of C such that C evaluates to one. ■

Remark V.8 *The formula constructed in the proof above is actually in* 3*-CNF. So we directly get that* 3SAT *is* NP*-complete.*

Exercise V.2 *Construct a similar expression as in (V.1) for the Boolean or.*

25 A universal Turing machine

25.1 Gödel numberings for Turing machines

We want to find a mapping that maps Turing machines to strings in $\{0,1\}^*$. Let $M = (Q, \{0,1\}, \Gamma, \delta, q_0, Q_{\text{acc}})$ be a k-tape Turing machine. We can assume that $Q = \{0, 1, \ldots, s\}$ and $\Gamma = \{0, 1, \ldots, \ell\}$. We assume that ℓ is the blank.

We can encode a state q by $\text{bin}(q)$ and a symbol γ by $\text{bin}(\gamma)$. The fact that

$$\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma'_1, \ldots, \gamma'_k, r_1, \ldots, r_k)$$

can be encoded by

$$[\text{bin}(q), \text{bin}(\gamma_1), \ldots, \text{bin}(\gamma_k), \text{bin}(q'), \text{bin}(\gamma'_1), \ldots, \text{bin}(\gamma'_k), \hat{r}_1, \ldots, \hat{r}_k]$$

where

$$\hat{r}_\kappa = \begin{cases} 00 & \text{if } r_\kappa = S \\ 10 & \text{if } r_\kappa = L \\ 01 & \text{if } r_\kappa = R \end{cases}$$

$[.,.]$ denotes (one of) the pairing functions discussed in Section 18.1.3. It is extended to larger tuples as expected:

$$[a_1, \ldots, a_m] := [a_1, [a_2, \ldots, [a_{m-1}, a_m]]].$$

If $\delta(q, \gamma_1, \ldots, \gamma_k)$ is undefined, then we encode this by

$$[\text{bin}(q), \text{bin}(\gamma_1), \ldots, \text{bin}(\gamma_k), \text{bin}(s+1), \varepsilon, \ldots, \varepsilon, \varepsilon, \ldots, \varepsilon]$$

The second part of the tuple is a dummy value, the non-existing state $s+1$ is used for saying that the value is undefined.

We construct a mapping göd_{TM} from the set of all Turing machines to $\{0,1\}^*$ by building a large pair consisting of:

- $\text{bin}(k)$, the number of tapes,
- $\text{bin}(s+1)$, the size of Q,
- $\text{bin}(\ell+1)$, the size of Γ,
- the encodings of $\delta(q, \gamma_1, \ldots, \gamma_k)$, $q \in Q$, and $\gamma_1, \ldots, \gamma_k \in \Gamma$, in lexicographic order,

Input: $i \in \{0,1\}^*$
Output: accept, if $i \in \operatorname{im} \text{göd}_{TM}$, reject otherwise

1. Extract the values k, s, and r from i.
2. From these values, C_{TM} can compute the size and the number of tuples encoding the transition function.
3. Test whether all these tupels are in ascending order, whether they have the correct number of entries, and whether the entries are all in the given bounds.
4. Finally check whether q_0 and whether the accepting states are between 0 and s and whether they are in ascending order and every state is only listed once.
5. If one of these conditions is violated, then reject. Otherwise accept.

Figure 25.1: The Turing machine C_{TM}.

- $\text{bin}(q_0)$, the start state,
- $\text{bin}(|Q_{\text{acc}}|)$, the number of accepting states,
- $\text{bin}(q)$, $q \in Q_{\text{acc}}$, in ascending order.

If M is supposed to compute a function (instead of recognizing some language $L \subseteq \{0,1\}^*$), then we indicate this by not giving any accepting states. It is clear that göd_{TM} is an injective mapping. In contrast to WHILE programs, göd_{TM} is not a bijection. This is no problem, since the image of göd_{TM} is decidable.

Constructing the Turing machine C_{TM} that computes the characteristic function of $\operatorname{im} \text{göd}_{TM}$ is rather easy. An index $i \in \operatorname{im} \text{göd}_{TM}$ is just a concatenation of some tupels, we just have to check whether they have the right form.

25.2 A universal Turing machine

Finally, we construct the universal Turing machine U_{TM}. U_{TM} simulates a given Turing machine M step by step. The biggest problem for constructing a universal Turing machine U_{TM} is that is has a fixed number of tapes, a

fixed working alphabet, and a fixed number of states. To simulate M, we encode the symbols of the work alphabet of M in binary. We store all k tapes of M on one tape of U_{TM}, the second one, say. To do so, we build blocks. If in some step, the ith cells of the k tapes of M contain the symbols $i_1, \ldots, i_k$, then the ith block is

$$\#\operatorname{bin}(i_1)\#\operatorname{bin}(i_2)\#\ldots\#\operatorname{bin}(i_k).$$

To the right of this block, there is the block corresponding to the $(i+1)$th cells of M, to left the one corresponding to the $(i-1)$th cells of M. Between two such blocks, U_{TM} writes \$ as a separator. So the k tapes of M are "interleaved".

Of course, U_{TM} has to bring its second tape into this form. In the beginning, it initializes its second tape by writing the blocks

$$\#\operatorname{bin}(x_j)\#\operatorname{bin}(\ell)\#\ldots\#\operatorname{bin}(\ell)$$

for $j = 1, \ldots, n$ on it where $x = x_1x_2\ldots x_n$ denotes the input for M. Recall the ℓ is the blank of M. Whenever M enters a cell that has not been visited before on any tape, then U_{TM} will enter a cell that contains a blank (of U_{TM}). U_{TM} then first creates a new block that consists solely of blanks (of M).

U_{TM} has only one head on its second tape. But M has k heads on its k tapes. U_{TM} remembers that the head of the jth tape of M is standing on the ith cell by replacing the jth # of the ith block by a $*$.

One can prove the correctness of U_{TM} by induction on the number of steps of M. This is not hard, but quite some slave work. We get the following result.

Theorem 25.1 *There is a Turing machine U_{TM} that, given a pair $[g, x]$ with $g \in \operatorname{im} \text{göd}_{TM}$ and $x \in \{0,1\}^*$, computes $\varphi_{\text{göd}_{TM}^{-1}(g)}(x)$.*

Exercise 25.1 *Show that the constructed Turing machine U_{TM} is correct.*

Input: $[g, x]$ with $g \in \operatorname{im} \text{göd}_{TM}$ and $x \in \{0,1\}^*$
Output: $\varphi_{\text{göd}_{TM}^{-1}(g)}(x)$

1. U_{TM} copies the start state to the third tape.
2. U_{TM} copies the input x from the first tape to the second tape as described above.
   It replaces all # of the first block by $*$.
3. U_{TM} moves the head to the first symbol of the leftmost block.
4. While the transition function is not undefined
   (a) U_{TM} goes to the right.
   (b) Whenever it finds a $*$, it copies the following number in binary to the fourth tape.
   (c) If U_{TM} reaches the right end of tape 2, then it looks up the tuple of δ that corresponds to the current state (on tape 3) and the symbols copied to tape 4.
   (d) U_{TM} replaces the state on tape 3 by the new state.
   (e) U_{TM} goes to the left
   (f) Whenever U_{TM} finds a $*$, it updates the corresponding cells and moves the $*$ to its new position.
5. If $M := \text{göd}_{TM}^{-1}(g)$ is supposed to compute a function, then U_{TM} copies the content of tape 2 that corresponds to the first tape of M back to tape 1 and stops.
6. If M is supposed to decide a language, then M accepts if the current state on tape 3 is in the list of accepting states of M, otherwise it rejects.

Figure 25.2: The universal Turing machine U_{TM}

26 Space and time hierarchies

Hierarchies

Is more space more power? Is more time more power?
The answer is "yes" provided that the space and time bounds behave well, that is, they shall be constructible.

In the case of time "more" means "somewhat more" and not just "more" (see Theorem 20.7).

26.1 A technical lemma

Lemma 26.1 *Let $s_1, s_2, t_1, t_2 : \mathbb{N} \to \mathbb{N}$ with $s_1 = o(s_2)$ and $t_1 = o(t_2)$. Assume that $s_2(n) \geq \log n$ and $t_2(n) \geq (1+\epsilon)n$ for all n and some $\epsilon > 0$. Let s_2 be space constructible and t_2 be time constructible.*

1. *There is a deterministic Turing machine C_1 that is s_2 space bounded such that for every s_1 space bounded 1-tape Turing machine M, $L(C_1) \neq L(M)$.*

2. *There is a deterministic Turing machine C_2 that is t_2 time bounded such that for every t_1 time bounded 1-tape Turing machine M, $L(C_2) \neq L(M)$.*

Proof overview: The proof is by diagonalization. When we constructed a function that is not WHILE (or Turing) computable, we just constructed a function whose value on i was different from $f_i(i)$. Here the function that we construct has to be computable in space s_2 and time t_2, respectively, which complicates the construction somewhat. We use the universal Turing machine to compute the values $f_i(i)$ and then diagonalize.

Proof. Let g be the Gödel number of some 1-tape Turing machine M. It is easy to modify the universal Turing machine in such a way that it can simulate a t-time bounded 1-tape Turing machine in time $O(|g| \cdot t(n))$ on inputs of length n:

- We use one tape to simulate M. The ith symbol of the work alphabet is represented by the string $\text{bin}(i)$ (fixed length binary encoding) and the symbols are separated by #.

(Remember that the tape alphabet of M might be much larger than the alphabet of U.)

- We do not need to mark the position of the head of M on this tape, since we can simply use the head of U on this tape.
- The encoding g of M stands on a second tape of U, the current state of M, stored as a number in binary, stands on the third tape of M.
- Now U simulates one step of M as follows. It looks in g for the entry that corresponds to the current state of M and the current symbol of M on the first tape. Then it replaces the state and the symbol and moves the head accordingly. In one such step, we have to read g once, hence simulating one step of M can be done by $O(g)$ steps of U.

Note that U is $O(|g| \cdot s(n))$ space bounded, when M is s space bounded. If M has an extra input tape, then U also has an extra input tape. (But this was even true for U_{TM}.)

We use U to construct C_1:

Input: $x \in \{0,1\}^*$, interpreted as $[g, y]$ with $g \in \operatorname{im} \text{göd}_{TM}$.

1. If x does not have the above form, then reject.
2. Mark $s_2(|x|)$ symbols to the left and right of cell 0 on the first tape.
3. Simulate $M := \text{göd}_{TM}^{-1}(g)$ on x on the first tape (using the machine U).
4. On an extra tape, count the number of simulated steps.
5. If the simulation ever leaves the marked cells, then stop and reject.
6. If more than $3^{s_2(|x|)}$ step are simulated, then stop and accept. (We can count up to this value by marking $s_2(|x|)$ cells and counting in ternary.)
7. Accept, if M rejects. Otherwise, reject.

Now let M be a s_1 space bounded 1-tape Turing machine. We claim that $L(M) \neq L(C_1)$. Let g be the Gödel number of M and let $x = [g, z]$ for some sufficiently long z.

First assume that $x \in L(C_1)$. We will show that in this case, $x \notin L(M)$, which means that $L(M) \neq L(C_1)$. If $x \in L(C_1)$, then C_1 accepts

x. This means that either M performed more than $3^{s_2(|x|)}$ many steps or M halts on x and rejects. In the second case, we are done. For the first case, note that there is a constant c such that M cannot make more than $c^{s_1(|x|)} \cdot (s_1(|x|)+2) \cdot (|x|+2)$ steps without entering an infinite loop.[1] Thus if $3^{s_2(|x|)} > c^{s_1(|x|)} \cdot (s_1(|x|)+2) \cdot (|x|+2)$ then we get that $x \notin L(M)$. But $3^{s_2(|x|)} > c^{s_1(|x|)} \cdot (s_1(|x|)+2) \cdot (|x|+2)$ is equivalent to

$$\log 3 \cdot s_2(|x|) > \log c \cdot s_1(|x|) + \log(s_1(|x|)+2) + \log(|x|+2)$$

This is fulfilled by assumption for all long enough x, i.e., for long enough z, because $s_2(|x|) \geq \log(|x|)$.

The second case is $x \notin L(C_1)$. We will show that now M accepts x. Note that C_1 always terminates. If C_1 rejects y, then M ran out of space or M halted and accepted. The second case, $x \in L(M)$ and we are done. We will next show that the first case cannot happen. Since M is s_1 space bounded, the simulation via U needs space $|g| \cdot s_1(|x|)$. But $|g| \cdot s_1(|x|) \leq s_2(|x|)$ for sufficiently large $|x|$. Thus this case cannot happen.

C_1 is s_2 space bounded by construction. This proves the theorem.

The construction of C_2 is similar, even easier. We do not have to check whether M runs out of space. We do not need to count to $3^{s_2(|x|)}$ to detect infinite loops. Instead we count the number of steps made by C_2. If more then $t_2(|x|)$ step are made, then we stop and reject. In this way, C_2 becomes $\mathrm{O}(t_2)$ time bounded. (To get down to t_2, use acceleration.) Since we can simulate one step of M by $|g|$ steps, the simulation of M takes $|g| \cdot t_1(|x|)$ steps of C_2 provided that M is t_1 time bounded. This is less than $t_2(|x|)$ if z is long enough. The rest of the proof is similar. ∎

26.2 Deterministic hierarchy theorems

Theorem 26.2 (Deterministic space hierarchy) *Let $s_2(n) \geq \log n$ be space constructible and $s_1(n) = \mathrm{o}(s_2(n))$. Then*

$$\mathsf{DSpace}(s_1) \subsetneq \mathsf{DSpace}(s_2).$$

Proof. Consider C_1 from Lemma 26.1. $L(C_1) \in \mathsf{DSpace}(s_2)$. There is no s_1 space bounded deterministic 1-tape Turing machine M such that $L(M) = L(C_1)$. But for every s_1 space bounded deterministic k-tape Turing machine N there is a s_1 space bounded deterministic 1-tape Turing machine N' with $L(N') = L(N)$ by Theorem 20.7. Thus there is also no s_1 space bounded deterministic k-tape Turing machine M such that $L(M) = L(C_1)$. ∎

[1] We cannot bound $|x|+2$ by $d^{s_1(|x|)}$, since s_1 might be sublogarithmic.

Next, we do the same for time complexity classes. The result will not be as nice as for space complexity, since we cannot simulate arbitrary deterministic Turing machines by 1-tape Turing machines without any slowdown.

Theorem 26.3 (Deterministic time hierarchy) *Let t_2 be time constructible and $t_1^2 = o(t_2)$. Then*

$$\mathsf{DTime}(t_1) \subsetneq \mathsf{DTime}(t_2).$$

Proof. Consider C_2 from Lemma 26.1. $L(C_2) \in \mathsf{DTime}(t_2)$. There is no t_1^2 time bounded deterministic 1-tape Turing machine M such that $L(M) = L(C_2)$. But for every t_1 space bounded deterministic k-tape Turing machine N there is a t_1^2 time bounded deterministic 1-tape Turing machine N' with $L(N') = L(N)$ by Theorem 20.7. Thus there is also no t_1 time bounded deterministic k-tape Turing machine M such that $L(M) = L(C_1)$. ∎

26.3 Remarks

The assumption $t_1^2 = o(t_2)$ in the proof of the time hierarchy theorem is needed, since we incurr a quadractic slowdown when simulating k-tape Turing machines by 1-tape Turing machines.

Hennie and Stearns showed the following theorem.

Theorem 26.4 (Hennie & Stearns) *Every t time and s space bounded deterministic k-tape Turing machine can be simulated by an $O(t \log t)$ time bounded and $O(s)$ space bounded deterministic 2-tape Turing machine.*

We do not give a proof here. Using this theorem, we let C_2 diagonalizes against 2-tape Turing machines instead of 1-tape Turing machines. This gives the following stronger version of the time hierarchy theorem.

Theorem 26.5 *Let t_2 be time constructible and $t_1 \log t_1 = o(t_2)$. Then*

$$\mathsf{DTime}(t_1) \subsetneq \mathsf{DTime}(t_2).$$

The answer to the following question is not known.[2]

Research Problem 26.1 *Can the assumption $t_1 \log t_1 = o(t_2)$ be further weakened? In particular, can we get a better simulation of arbitrary determinstirc Turing machines on Turing machines with a fixed number of tapes?*

If the number of tapes is fixed, then one can obtain a tight time hierarchy. Again we do not give a proof here.

[2] If you can answer it, we should talk about your dissertation.

Theorem 26.6 (Fürer) *Let $k \geq 2$, t_2 time constructible, and $t_1 = \mathrm{o}(t_2)$. Then*

$$\mathsf{DTime}_k(t_1) \subsetneq \mathsf{DTime}_k(t_2).$$

We conclude with pointing out that the assumption that s_2 and t_2 are constructible are really necessary.

Theorem 26.7 (Borodin's gap theorem) *Let f be a total recursive function $\mathbb{N} \to \mathbb{N}$ with $f(n) \geq n$ for all n. Then there are total recursive functions $s, t : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq n$ and $t(n) \geq n$ for all n such that*

$$\mathsf{DTime}(f(t(n))) = \mathsf{DTime}(t(n)),$$
$$\mathsf{DSpace}(f(s(n))) = \mathsf{DSpace}(s(n)).$$

Proof. We only construct t, the construction of s is similar. Let $T_g(n) := \mathrm{Time}_{\mathrm{göd}_{TM}^{-1}(g)}(n)$ be the maximum running time of the Turing machine with Gödel number g on inputs of length n.

We first show: For all $n \in \mathbb{N}$, there is an $m \in \mathbb{N}$ such that for all Gödel numbers g with $\mathrm{cod}(g) \leq n$,

$$T_g(n) \leq f(m) \Rightarrow T_g(n) \leq m.$$

Let $m_0 = n$ and $m_{i+1} = f(m_i) + 1$ für $1 \leq i \leq n+1$. The $n+2$ intervals $[m_i, f(m_i)]$ are pairwise disjoint, because $f(n) \geq n$ for all n. Therefore, there is an i_0 such that $T_g(n) \notin [m_{i_0}, f(m_{i_0})]$ for all g with $\mathrm{cod}(g) \leq n$. Set $m = m_{i_0}$.

Let $t(n)$ be the m defined above corresponding to n. t is recursive: We can compute the intervals, since f is total and recursive. We can test $T_g(n) \notin [m_i, f(m_i)]$ by simulating $\mathrm{göd}^{-1}(g)$ on all inputs of length n. Since each of these simulation can be stopped after $f(m_i)$ steps, this is decidable.

Now let M be $f(t(n))$ time bounded, i.e., $T_g(n) \leq f(t(n))$ for all n, where $g = \mathrm{göd}_{TM}(M)$. By the construction of t, $T_g(n) \leq t(n)$ for all $n \geq \mathrm{cod}(g)$. Therefore, $L(M) \in \mathsf{DTime}(t)$. Thus, $\mathsf{DTime}(f \circ t) = \mathsf{DTime}(t)$. ∎

Set for instance $f(n) = 2^n$ (or 2^{2^n} or ...) and think for a minute how unnatural non-constructible time or space bounds are.

Excursus: Nondeterministic hierarchies

For nondeterministic space, we can use Savitch's theorem to show the following:

$$\mathsf{NSpace}(s_1) \subseteq \mathsf{DSpace}(s_1^2) \subsetneq \mathsf{DSpace}(s_2^2) \subseteq \mathsf{NSpace}(s_2^2)$$

for any functions $s_1, s_2 : \mathbb{N} \to \mathbb{N}$ with $s_1 = \mathrm{o}(s_2)$, s_1 and s_2 space constructible, and $s_1(n) \geq \log n$. It is even possible to show a tight hierarchy like in Theorem 26.2. This uses the non-trivial—and unexpected—fact that $\mathsf{NSpace}(s)$ is closed under complementation, the so-called *Immerman–Szelepcsényi Theorem*.

For nondeterministic time, neither of the two approaches is known to work. But one can get the following hierarchy result: For a function $t : \mathbb{N} \to \mathbb{N}$, let $\tilde{t}$ be the function defined by $\tilde{t}(n) = t(n+1)$. If t_1 is time constructible and $\tilde{t}_1 = o(t_2)$ then

$$\mathsf{NTime}(t_2) \setminus \mathsf{NTime}(t_1) \neq \emptyset.$$

The proof of this result is lengthy. Note that for polynomial functions or exponential functions, $\tilde{t}_1 = O(t_1)$. Thus we get a tight nondeterministic time hierarchy for these functions.

27 More on NP

We conclude this part with some more stories about NP.

27.1 NP versus co-NP

For a complexity class C, co-C denotes the set of all languages $L \subseteq \Sigma^*$ such that $\bar{L} \in \mathsf{C}$. Deterministic complexity classes are usually closed under complementation, for instance, $\mathsf{DTime}(t) = \text{co-}\mathsf{DTime}(t)$ for any t and $\mathsf{P} = \text{co-P}$. For nondeterminsitc time complexity classes, it is a big open problem whether a class equals its co-class, in particular, whether $\mathsf{NP} = \text{co-NP}$. Note that we cannot just exchange accepting and nonaccepting states of a polynomial time bounded nondeterministic Turing machine M: If M does not accept x, then it will accept x after the exchange. This is fine. But if M accepts x, then it may still accept x after the exchange, since there might be rejecting computation paths of M on x that will become accepting by the exchange.

Let UNSAT be the encodings of all formulas in CNF that are not satisfiable. Note that UNSAT is *not* the complement of SAT. The complement of SAT is UNSAT together with all strings that are not an encoding of a formula in CNF. But since such strings can be recognized in polynomial time, we get that $\overline{\text{SAT}} \leq_{\text{P}}$ UNSAT. Since SAT is NP-complete, $\overline{\text{SAT}}$ is co-NP-complete (see the next Exercise) and so is UNSAT.

Exercise 27.1 *Show the following: If $L \leq_{\text{P}} L'$, then $\bar{L} \leq_{\text{P}} \bar{L}'$ (cf. Exercise 13.1).*

TAUT is the following problem: Given a formula in disjunctive normal form (in DNF for short), decide whether it is a tautology, i.e., whether all assignments satisfy it.

Exercise 27.2 *Show that* TAUT *is* co-NP*-complete.*

One approach to show that $\mathsf{P} \neq \mathsf{NP}$ would be to show that NP is not closed under complementation, i.e., $\mathsf{NP} \neq \text{co-NP}$. To show that NP is closed under complementation, it is sufficient to show that one NP-complete problem is in co-NP.

Theorem 27.1 *If* co-NP *contains an* NP*-complete problem, then* $\mathsf{NP} =$ co-NP*.*

Proof. Let $L \in$ co-NP be NP-complete.

Let $A \in$ NP arbitrary. Since L is NP-complete, there is a polynomial time many one reductions f from A to L. But since $L \in$ co-NP, A is also in co-NP: We first compute $f(x)$ and then run the nondeterministic machine for L. Thus NP $\subseteq$ co-NP.

Let $B \in$ co-NP. If L is NP-complete, then $\bar{L}$ is co-NP-complete, since a many one reduction f from some A to L is also a many one reduction from $\bar{A}$ to $\bar{L}$. A similar argument as above now shows that $B \in$ NP. ∎

A natural co-NP-complete problem is UNSAT, another one is TAUT. But we do not know whether they are in NP or not. Most researchers conjecture that they are not.

What is the relation between P and NP ∩ co-NP? PRIMES, the problem whether a given number (in binary) is a prime number, was *the* example of an interesting language in NP∩co-NP that is not known to be in P. Recently, Agarwal, Kayal, and Saxena (Annals of Mathematics, 160(2), 781-793, 2004) showed that PRIMES ∈ P.

FACTOR is another problem that is in NP ∩ co-NP but not known to be in P: Given two numbers x and c in binary, decide whether x has a factor b with $2 \leq b \leq c$.

Exercise 27.3 *Prove the following:*

1. FACTOR ∈ NP.
2. FACTOR ∈ co-NP. *(You can use that* PRIMES ∈ P*)*

At the moment, the most prominent example for a problem in NP∩co-NP is PARITY GAMES, an important problem that occurs in verification, but it is beyond the scope of this lecture ...

27.2 Self-reducibility

Assume we have a polynomial time deterministic algorithm for SAT. Then given a formula ϕ, we can find out whether it is satisfiable or not in polynomial time. But what we really want is a satisfying assignment.

SAT has a nice property, we can reduce questions about a formula ϕ in CNF to questions about smaller formulas. Let x be a variable of ϕ. Let ϕ_0 and ϕ_1 be the two formulas that are obtained by setting x to 0 or 1, respectively, and then removing clauses that are satisfied by this and removing literals that became 0. (This procedure can produce an empty clause. Such a formula is not satisfiable by definition. Or the procedure could produce the empty formula in CNF. This one is satisfiable.) Then ϕ is satisfiable if and only if ϕ_0 or ϕ_1 is satisfiable.

If we have an algorithm that decides SAT in polynomial time, say in time $p(\ell)$, then we can find a satisfying assignment recursively. Let ϕ be a satisfiable formula. If ϕ has only one variable, then finding a satisfying assignment is easy. If ϕ has more variables, then we test whether ϕ_0 is satisfiable and if it is, we compute a satisfying assignment for ϕ_0. This assignment together with setting x to 0 is a satisfying assignment for ϕ. If ϕ_0 is not satisfiable, then ϕ_1 is satisfiable, since ϕ is. We recursively compute a satisfying assignment for ϕ_1. This assignment together with setting x to 1 is a satisfying assignment for ϕ.

Note that the number of variables of ϕ_0 and ϕ_1 is smaller than the number of variables in ϕ. The recursion depth is therefore bounded by ℓ, the length of the formula. In each stage, there is only one recursive call and one evocation of the decision prodecure for SAT. Thus the total running time is $O(\ell \cdot p(\ell))$.

Exercise 27.4 *Assume that* TSP $\in$ P.

1. *Describe a polynomial time algorithm that given a complete weighted graph G computes the weight b of a minimum weight Hamiltonian tour. (Hint: binary search)*

2. *Describe a polynomial time algorithm that given a complete weighted graph G computes a minimum weight Hamiltonian tour. (Hint: self-reducibility)*

27.3 Approximation algorithms

While we do not know how to solve NP-hard problems efficiently, they occur frequently in practise and we have to "solve" them. Often, one does not need to solve a problem exactly but we are content with an approximate solution.

Let us illustrate this with the problem TSP. If there is a polynomial time deterministic Turing machine M that given a complete weighted graph G outputs a minimum weight Hamiltonian tour, then P = NP, because given the minimum weight Hamiltonian tour, we can decide whether a given pair (G, b) is in TSP or not.

But can we, say, compute a Hamiltonian tour whose weight w is at most twice the weight of a minimum weight Hamiltonian tour? In general, this it not possible. For an edge weighted graph G, let $\mathrm{OPT}(G)$ be the weight of a minimum Hamiltonian tour of G.

Theorem 27.2 *If there is polynomial time bounded deterministic Turing machine A that given an edge weighted graph $G = (V, \binom{V}{2}, w)$ with n nodes, computes a Hamiltonian tour H with $w(H) < 2^{p(n)} \cdot \mathrm{OPT}(G)$ for some polynomial p, then* P = NP.

Proof. We use essentially the same reduction as in Lemma 23.14. But instead of mapping the "non-edges" to 2, we map them to the value $n \cdot 2^{p(n)}$. This is still a polynomial time reduction, since we need only $p(n) + \log n$ bits to write down these values.

Now assume that the graph G has a Hamiltonian cycle. Then H has a Hamiltonian cycle of weight n. If G does not have a Hamiltonian cycle, then every Hamiltonian cycle of H has at least weight $n \cdot 2^{p(n)}$, since it contains at least one edge with this weight.

This means that if G has a Hamiltonian cycle, then A on H will return a Hamiltonian tour with weight $< n \cdot 2^{p(n)}$. If G does not have a Hamiltonian cycle, then A on H can only return a Hamiltonian tour of weight at least $n \cdot 2^{p(n)}$. This way, we can distinguish whether G has a Hamiltonian cycle or not. This gives a polynomial time deterministic Turing machine for HC, hence P = NP. ∎

But if w fulfills the triangle inequality, that is,

$$w(\{u,v\}) \leq w(\{u,x\}) + w(\{x,v\})$$

for all nodes $u, v, x \in V$, then we can compute approximate solutions.

Exercise 27.5 *Let $G = (V, \binom{V}{2}, w)$ be an edge weighted graph such that w fulfills the triangle inequality. Let T be a minimum spanning tree of G. Start in some node and do a depth first search on T. Order the nodes by the time they are visited first in this depth first search. Then the Hamiltonian tour that corresponds to this order hast weight $\leq 2 \cdot \mathrm{OPT}(G)$.*

27.4 Strongly NP-hard problems and pseudo-polynomial problems

A *number problem* is a language whose elements are (encodings of) sequences of natural or rational numbers. Subset-Sum is a number problem but also TSP is, since we just have to write down the values of the weight function. There are two natural ways to write down the instances of number problems: in binary or in unary.

Definition 27.3 *A number problem is* strongly NP-hard *if it is NP-hard when the numbers are written in unary.*[1]

[1] This is a somewhat sloppy definition but you find it in many textbooks. We have two languages: L_b, where the numbers are written in binary and L_u with the number written in unary. Strongly NP-hard means that L_u is NP-hard. (This implies of course that L_b is also NP-hard, since there is an easy reduction from L_b to L_u. The converse is not clear, since the obvious reduction is not polynomial time computable, as the length of the output might be exponential.)

By Exercise 23.2, there is a deterministic Turing machine for Subset-Sum with running time polynomial in n and b. This is called a *pseudo-polynomial* running time: the running time is polynomial in the length of the vector and the maximum size of the entries.[2]

Theorem 27.4 *If a strongly* NP*-hard number problem L is decidable by a pseudo-polynomially time bounded deterministic Turing machine, then* $\mathsf{P} = \mathsf{NP}$.

Proof. Let $A \in \mathsf{NP}$. There is a polynomial time many one reduction from A to L where the instances of L are written in unary. Since there is a pseudo-polynomially time bounded deterministic Turing machine for L and the elements of the instances are written in unary, $L \in \mathsf{P}$. Thus $A \in \mathsf{P}$. ∎

This means that Subset-Sum cannot be strongly NP-hard unless $\mathsf{P} = \mathsf{NP}$. On the other hand, we have seen that if the numbers are encoded in binary, then Subset-Sum is NP-hard.

TSP is unconditionally strongly NP-hard. The reduction in Lemma 23.14 only uses the numbers 1 and 2. Since every weight function with weights 1 and 2 fulfills the triangle inequality, even TSP with triangle inequality is strongly NP-hard.

Problems with pseudo-polynomial algorithms can often be approximated well. Consider the following version of Subset-Sum: Given $(x_1, \ldots, x_n, b)$, find an I such that $\sum_{i \subseteq I} x_i$ is as large as possible but not larger than b.[3] This problem also has a pseudo-polynomial algorithm. We can always assume that each x_i non-zero and not larger than b. Let OPT denote the maximal sum of x_i's that is not larger than b.

We will now show how we can, given an $\epsilon > 0$, compute in time polynomial in n and $1/\epsilon$ a set J such that $(1-\epsilon)\,\mathrm{OPT} \leq \sum_{j \in J} x_j \leq (1+\epsilon)\,\mathrm{OPT}$. This means that our solution can get arbitrarily close to the optimum (at the expense of a smaller ϵ). But since OPT can be as large as b, there is also the risk that we overpack slightly.

Let $x = \max\{x_1, \ldots, x_n\}$ and $L = \epsilon \cdot x/n$. Let $y_i = \lfloor x_i/L \rfloor$ and $c = \lfloor b/L \rfloor$. If we now run the pseudo-polynomial algorithm on this modified instance $(y_1, \ldots, y_n, c)$, then the overall running time is polynomial in $1/\epsilon$ and n. We get a set J such that $\sum_{j \in J} y_j \leq c$ and no other set is achieves a larger weight $\leq c$.

Let O be a set such that $\sum_{i \in O} x_i = \mathrm{OPT}$. By the optimality of J, $\sum_{i \in O} y_i \leq \sum_{j \in J} y_j$. This implies

$$(1-\epsilon)\,\mathrm{OPT} \leq \sum_{i \in O} x_i - L \cdot n \leq L \cdot \sum_{i \in O} y_i \leq L \cdot \sum_{j \in J} y_j \leq \sum_{j \in J} x_j,$$

[2] Using the notation of the previous footnote, this means that $L_u \in \mathsf{P}$.

[3] A problem that is very handy in a Swedish furniture store: You are interested in goods with weights $x_1, \ldots, x_n$, but your car can only carry load b ...

since $\mathrm{OPT} \geq x$ and $x_k - L \leq L \cdot y_k \leq x_k$ for all k. On the other hand

$$\sum_{j \in J} x_j \leq L \sum_{j \in J} y_j + nL \leq (1+\epsilon)\,\mathrm{OPT}\,.$$

Excursus: WHILE programs and unit costs

WHILE programs with unit costs and more operations than our simple statements can be really powerful. Assume that we can perform multiplication and division with remainder with unit costs.

Let $y = (x_1, \ldots, x_n, b)$ be an instance of Subset-Sum. Let $x = \max\{x_1, \ldots, x_n\}$. Consider the product

$$P := (1 + 2^{Z \cdot x_1}) \cdots (1 + 2^{Z \cdot x_n}) = \sum_{S \subseteq \{1,\ldots,n\}} 2^{Z \cdot \sum_{i \in S} x_i} = \sum_m c_m 2^{Z \cdot m},$$

where c_m is the number of sets S such that $\sum_{i \in S} x_i = m$. $c_m \leq 2^n$ since there are at most 2^n sets. Thus if we set $Z = n + 1$, then we have no "carries" and get that $y \in$ Subset-Sum iff $c_b \neq 0$.

We can compute c_b by $P / 2^{Z \cdot (b+1)} \operatorname{rem} 2^Z$ where "/" denotes the integer division and "rem" computes the remainder.

Under unit costs, the number of operations is polynomial in n and $\log x$, which is the size of the instance. To compute $2^{Z \cdot x_i}$, we first compute 2^Z and then $(2^Z)^{x_i}$. The latter only needs $\log x_i$ multiplications by using the square and multiply technique: $2 \mapsto 2^2 = 4 \mapsto 4^2 = 16 \mapsto \ldots$.

On the other hand, note that under logarithmic costs, we do not have polynomial running time, since P has more than $Z \cdot x$ bits.

Part IV

Grammars

28 Grammars

When Noam Chomsky invented grammars, he wanted to study sentences in natural languages. He wanted to formulate rules like a *Satz* in German consists of a *Subjekt* followed by a *Prädikat* and then maybe followed by an *Objekt*. A *Subjekt* consists of an *Artikel* and a *Nomen*. An *Artikel* can be *der*, *die*, or *das*. A lot of words can be a *Nomen*, examples are *Hund*, *Katze*, and *Maus*.[1]

Definition 28.1 *A grammar G is described by a 4-tuple (V, Σ, P, S).*

1. *V is a finite set, the set of* variables *or* nonterminals.
2. *Σ is a finite set, the set of* terminal symbols. *We have $V \cap \Sigma = \emptyset$.*
3. *P is a finite subset of $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$, the set of* productions.
4. *$S \in V$ is the* start variable.

Convention 28.2 *If $(u, v) \in P$ is a production, we will often write $u \to v$ instead of (u, v).*

In the example above, *Satz* would be the start variable. *Subjekt*, *Prädikat*, ... would be variables. The letters "d", "e", "r", ... are terminal symbols.

Definition 28.3 1. *A grammar $G = (V, \Sigma, P, S)$ defines a relation $\Rightarrow_G$ on $(V \cup \Sigma)^*$ as follows: $u \Rightarrow_G v$ if we can write $u = xyz$ and $v = xy'z$ and there is a production $y \to y' \in P$. We say that v is* derivable *from u* in one step. *v is* derivable *from u if $u \Rightarrow_G^* v$, where $\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$.*

2. *A word $u \in (V \cup \Sigma)^*$ is called a* sentence *if $S \Rightarrow_G^* u$.*
3. *A sequence of sentences $w_0, \ldots, w_t \in (V \cup \Sigma)^*$ with $w_0 = S$, $w_\tau \Rightarrow_G w_{\tau+1}$ for $0 \leq \tau < t$, and $w_t = u$ is called a* derivation *of u. (A derivation can be considered as a* witness *or* proof *that $S \Rightarrow_G^* u$.)*
4. *The language generated by G is $L(G) = \{u \in \Sigma^* \mid S \Rightarrow_G^* u\}$. (Note that words in $L(G)$ do not contain any variables.)*

[1] Yes, I know, this is oversimplified and not complete.

Example 28.4 *Let* $G_1 = (\{S\}, \{0,1\}, P_1, S)$ *where* P_1 *consists of the productions*

$$S \to \varepsilon$$
$$S \to 0S1$$

Exercise 28.1 *Prove by induction on* i *that* $0^i S 1^i$ *is the only sentence of length* $2i+1$ *with* $S \Rightarrow_{G_1}^* 0^i S 1^i$. *Conclude that* $L(G_1) = \{0^i 1^i \mid i \in \mathbb{N}\}$.

Syntactic sugar 28.5 *If* P *contains the productions* $u \to v_1, \ldots, u \to v_t$ *(with the same left-hand sides* u*), then we will often write* $u \to v_1 \mid \cdots \mid v_t$.

Example 28.6 *Let* $G_2 = (\{W, V, N, N^+, Z, Z^+\}, \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}, \mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}, :, =, \neq, ;, +, -, [,]\}, P_2, W)$ *where* P_2 *consists of the productions*

$$\begin{aligned}
W &\to V{:=}V{+}V \mid V{:=}V{-}V \mid V{:=}N^+ \mid \\
&\quad \texttt{while } V{\neq}\texttt{ 0 do } W \texttt{ od} \mid \\
&\quad [W;W] \\
V &\to \mathtt{x}N^+ \\
N^+ &\to Z \mid Z^+N \\
N &\to Z \mid ZN \\
Z &\to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
Z^+ &\to \mathtt{1} \mid \mathtt{2} \mid \cdots \mid \mathtt{9}
\end{aligned}$$

It is quite easy (but a little tedious) to see that $L(G_2)$ *is the set of all WHILE programs (now over a finite alphabet).*[2] *From* N^+*, we can derive all decimal representations of natural numbers (without leading zeros). From* V*, we can derive all variable names. From* W*, we can derive all WHILE programs. The first three productions produce the simple statements, the other two the while loop and the concatenation.*

Example 28.7 *Let* $G_3 = (\{S, E, Z\}, \{0,1,2\}, P_3, S)$ *where* P_3 *is given by*

$$\begin{aligned}
S &\to 0EZ \mid 0SEZ \\
ZE &\to EZ \\
0E &\to 01 \\
1E &\to 11 \\
1Z &\to 12 \\
2Z &\to 22
\end{aligned}$$

[2] Because of the simple structure of WHILE programs, we do not even need whitespaces to separate the elements. Feel free to insert them if you like.

Exercise 28.2 *1. Prove by induction on n that $S \Rightarrow_{G_3}^* 0^n1^n2^n$ for all $n \geq 1$.*

2. Show that whenever $S \Rightarrow_{G_3}^* w$, then the number of 0's in w equals the number of E's plus 1's in w and it also equals the number of Z's plus 2's in w.

3. Show that whenever one uses the rule $1Z \to 12$ and there is an E to the right of the 2 created, then one cannot derive a word from Σ^* from the resulting sentence.

4. Conclude that $L(G_3) = \{0^n1^n2^n \mid n \geq 1\}$.

28.1 The Chomsky hierachy

Definition 28.8 *Let $G = (V, \Sigma, P, S)$ be a grammar.*

1. *Every grammar G is a* type-0 grammar.
2. *G is a* type-1 grammar *if $|u| \leq |v|$ for every production $u \to v \in P$. The only exception is the production $S \to \varepsilon$. If $S \to \varepsilon \in P$, then S does not appear in the right-hand side of any production of P.*
3. *G is a* type-2 grammar *if it is type-1 and in addition, the left-hand side of every production is an element from V.*
4. *G is a* type-3 grammar *if it is type-2 and in addition, the right-hand side of every production is of the form $\Sigma V \cup \Sigma$, except for a potential production $S \to \varepsilon$.*

Definition 28.9 *Let $i \in \{0, 1, 2, 3\}$. A language $L \subseteq \Sigma^*$ is a* type-i language *if there is a type-i grammar G with $L = L(G)$.*

The grammar in the Example 28.6 is a type-2 grammar. Type-2 grammars are also called *context-free grammars* and type-2 languages are called *context-free languages*. The idea behind this name is that we can replace a variable A in a sentence regardless of the context it is standing in.

Type-1 grammars are also called *context-sensitive grammars* and type-1 languages are called *context-sensitive languages*. Here rules of the form $xAy \to w$ are possible and we can replace A only if it stands in the *context* xAy.[3]

Type-3 grammars are also called *right-linear grammars*: "linear", because the derivation trees (to be defined in the next chapter) degenerate essentially

[3]The name context-sensitive is not too well chosen, type-0 grammars have the same property, too, since they are more general. Nevertheless, the term context-sensitive is reserved for type-1 grammars and languages. But the important property of type-1 grammars is that they cannot shorten sentences by applying a production.

to a linear chain, and "right" because the variable stand at the right-hand end of the productions. Type-3 grammars are called *regular grammars*, too. Theorem 28.12 explains this: type-3 languages are exactly the regular languages. The grammar that we get from the grammar in Example 28.6 by only taking the variables $\{N, N^+, Z, Z^+\}$, these productions that use the variables $\{N, N^+, Z, Z^+\}$, and the start symbol N^+ generates the digital representations without leading zeros of all natural numbers. It is "almost" right linear. The variables Z and Z^+ are just place holders for a bunch of terminals. The grammar gets right linear if we replace the productions of the type $N \to ZN$ and $N^+ \to Z^+N$ by productions of the form $N \to 0N$, $N \to 1N$, etc.

Definition 28.10 *1. The set of all type-2 languages is denoted by* CFL.

2. The set of all type-1 languages is denoted by CSL.

By definition, the set of all type-3 languages is a subset of all type-2 languages. This inclusion is strict, since $\{0^n1^n \mid n \in \mathbb{N}\}$ is context-free but not regular. The grammar G_1 in Example 28.4 is an "almost context-free" grammar for this language. The only problem is that we can derive ε from the start symbol S and S is standing on the righthand side of some productions. But for context-free grammars, this is not a real problem. The grammar

$$S \to \varepsilon \mid S'$$
$$S' \to 01 \mid 0S'1$$

is a type-2 grammar for $\{0^n1^n \mid n \in \mathbb{N}\}$. We will later see a general way to get rid of productions of the form $A \to \varepsilon$ in an "almost context-free" grammar. (Note that this is not possible for context-sensitive grammars!)

Syntactic sugar 28.11 *When we write down a context-free grammar, we often only write down the productions in the following. Then the following conventions apply:*

1. *The symbols on the lefthand side are the variables.*
2. *All other symbols are terminals.*
3. *The lefthand side of the first productions is the start variable.*

In the same way, type-2 languages are a subset of the type-1 languages. The language $\{0^n1^n2^n \mid n \geq 1\}$ is context-sensitive, as shown in Example 28.7, but we will see soon that it is not context-free. Hence this inclusion is also strict.

The set of all type-0 languages equals RE—a fact that we will not prove here. On the other hand, CSL $\subseteq$ REC: Given a string $w \in \Sigma^*$, we can

generate all derivations for words of length $|w|$, because once we reached a sentence of length $> |w|$ in the derivation, we can stop, since productions of context-sensitive grammars can never shorten a sentence. Thus the type-1 languages are a strict subset of the type-0 languages.

28.2 Type-3 languages

Theorem 28.12 *Let $L \subseteq \Sigma^*$. L is a type-3 language iff L is regular.*

Proof. "$\Rightarrow$": Let $G = (V, \Sigma, P, S)$ be a type-3 grammar with $L(G) = L$. We will construct a nondeterministic finite automaton M with $L(M) = L$. We first assume that $\varepsilon \notin L(G)$. Let $F \notin V$. We set $M = (V \cup \{F\}, \Sigma, \delta, S, \{F\})$ where

$$\delta(A, \sigma) = \begin{cases} \{B \mid A \to \sigma B \in P\} & \text{if } A \to \sigma \notin P \\ \{B \mid A \to \sigma B \in P\} \cup \{F\} & \text{otherwise.} \end{cases}$$

Let $w = w_1 w_2 \dots w_n \in \Sigma^* \setminus \{\varepsilon\}$. We have

$$\begin{aligned} w \in L(G) &\iff \text{there are variables } V_1, \dots, V_{n-1} \in V \text{ with} \\ &\qquad S \Rightarrow_G w_1 V_1 \Rightarrow_G w_1 w_2 V_2 \Rightarrow_G \dots \Rightarrow_G w_1 w_2 \dots w_{n-1} V_{n-1} \Rightarrow_G w_1 w_2 \dots w_{n-1} w_n \\ &\iff \text{there are states } V_1, \dots, V_{n-1} \in V \text{ with} \\ &\qquad V_1 \in \delta(S, w_1), V_2 \in \delta(V_1, w_2), \dots, V_{n-1} \in \delta(V_{n-2}, w_{n-1}), F \in \delta(V_{n-1}, w_n) \\ &\iff w \in L(M). \end{aligned}$$

If $\varepsilon \in L(G)$, then we first construct an automaton M for $L(G) \setminus \{\varepsilon\}$ first.[4] Then we add a new start state that is also an end state and add an ε-transition to the old start state of M. This automaton recognizes $L(M) \cup \{\varepsilon\} = L(G)$.
"$\Leftarrow$": is shown in Exercise 28.3. ∎

Exercise 28.3 *Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a deterministic finite automaton. Let $G = (Q, \Sigma, P, q_0)$ where the set of productions contains a production*

$$q \to \sigma q' \quad \text{for all } q, q' \in Q \text{ and } \sigma \in \Sigma \text{ with } \delta(q, \sigma) = q'$$

and in addition

$$q \to \sigma \quad \text{for all } q \in Q,\ q' \in Q_{\text{acc}}, \text{ and } \sigma \in \Sigma \text{ with } \delta(q, \sigma) = q'.$$

If $q_0 \in Q_{\text{acc}}$, we will add the production $q_0 \to \varepsilon$, too. In this case, q_0 is not allowed to appear on the right-hand side of a production. If this happens, then we modify the automaton M by copying the start state. Show that $L(G) = L(M)$.

[4] This is possible, since we get a type-3 grammar for $L(G) \setminus \{\varepsilon\}$ by removing the production $S \to \varepsilon$.

Type-3 languages (regular languages)

Let $L \subseteq \Sigma^*$. The following statements are equivalent:

- There is a right-linear grammar G with $L(G) = L$.
- There is a deterministic finite automaton M with $L(M) = L$.
- There is a nondeterministic finite automaton M with $L(M) = L$.
- There is a regular expression E with $L(E) = L$.
- $\sim_L$ has finite index.

Although all these concepts describe regular languages, they have different properties: nondeterministic finite automata often have much fewer states than deterministic ones for the same language. Deciding whether two deterministic finite automata recognize the same language is easy whereas this is a hard problem for regular expressions (we will see this later on), etc.

Exercise 28.4 *A grammar $G = (V, \Sigma, P, S)$ is called* left-linear *if it is type-2 and the right-hand sides of all productions are of the form $V\Sigma \cup \Sigma$, except for a potential production $S \to \varepsilon$. Let $L \subseteq \Sigma^*$. Show that there is a left-linear grammar G with $L(G) = L$ iff L is regular.*

VI Grammars versus Turing machines

In this chapter, we investigate the relation between type-0 grammars and Turing machines as well as between type-1 grammars and a restricted class of Turing machines, so-called linear bounded automata.

VI.1 Type-0 versus RE

Lemma VI.1 *For every nondeterministic Turing machine N, there is a deterministic Turing machine M such $L(N) = L(M)$.*

Remark VI.2 *We showed this result under the additional assumption that N is weakly t time bounded for some time constructible t. Here we even do not know an upper bound on the running time of N.*

Proof. M acts as follows:

Input: x

1. Set $t = 1$;
2. Simulate N deterministically on x cutting off every computation path at length t.
3. If N accepts on one path, then accept, too.
4. Otherwise, double t and go to step 2.

If N accepts x, then there is an accepting path of some length t_0. When $t \geq t_0$, M will find this path and accept. If N does not accept x, then M will not find any accepting path and never stop and henceforth not accept x. ■

Lemma VI.3 *Let $G = (V, \Sigma, P, S)$ be a type-0 grammar. Then there is a Turing machine M such that $L(G) = L(M)$.*

Proof. By the lemma above, it suffices to construct a nondeterministic Turing machine M.

Input: $x \in \Sigma^*$

1. Write S on the work tape.
2. Nondeterministically choose some production from P and some position in the sentence on the work tape and apply the production if possible.
3. Check whether the sentence is x. If yes, accept. If not, goto step 2.

If $x \in L(G)$, then there is some derivation for x. By construction, the Turing machine M will find this derivation along some computation path. If $x \notin L(G)$, then M will never stop on any computation path. ■

Lemma VI.4 *Let M be a nondeterministic Turing machine. Then there is a type-0 grammar G such that $L(M) = L(G)$.*

Proof overview: If M accepts x, then $\mathrm{SC}(x) \vdash C_1 \vdash \ldots \vdash C_t$ for configurations $C_1, \ldots, C_t$ and C_t is an accepting configuration. If we write down the C_τ in a proper way, then C_τ and $C_{\tau+1}$ differ only locally. G can simulate this transition by an appropriate production. The only problem is that G has to generate x, but M might erase x during this computation. Therefore, our symbols have two components and we use the first just for storing x and second for the simulation of M.

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_{\mathrm{acc}})$ be a nondeterministic Turing machine. We can assume that M has only one tape that is onesided infinite. The end of this tape is marked by the symbol \$. Furthermore, we can assume that M moves its head in every step and that M immediately stops, whenever it enters an accepting state.

Our grammar $G = (V, \Sigma, P, S)$ looks as follows. We first have a set of "starting productions":

$$\begin{array}{rll} S & \to (\varepsilon, \$)T & \\ T & \to (\sigma, \sigma)T & \text{for all } \sigma \in \Sigma \\ T & \to U & \\ U & \to (\varepsilon, \square)U & \\ U & \to R & \\ AR & \to RA & \text{for all } A \in V \\ R(\varepsilon, \$) & \to (\varepsilon, \$)q_0 & \end{array}$$

The sentences that we can produce with these rules are of the form

$$(\varepsilon, \$)q_0(\sigma_1, \sigma_1)\dots,(\sigma_n, \sigma_n)(\varepsilon, \square)^i$$

for all $n, i \in \mathbb{N}$. The second component of the variables are used to simulate the Turing machine. Its concatenation always represents the content of the tape. In the beginning, it is $\$\sigma_1 \dots \sigma_n \square^i$, which is the tape content when starting the computation on $\sigma_1 \dots \sigma_n$. The variable q_0 (obviously) stores the state and also marks the position of the head. The convention is that the head of the Turing machine is always on the symbol to the right of the state. The first component is just used to store the input word, since in the end, the grammars has to produce the word while the Turing machine only has to accept or reject it and potentially could have erased the input. The variable R is only neccessary to create a nice "interface" to the second part of the grammar; productions of the second part can only be applied after we generated a sentence as shown above.

$$\begin{array}{rll} q(\sigma,\gamma) & \to (\sigma,\gamma')q' & \text{for all } q \in Q,\ \sigma \in \Sigma \cup \{\varepsilon\},\ \gamma \in \Gamma, \\ & & \text{and } (q',\gamma',R) \in \delta(q,\gamma) \\ (\tau,\beta)q(\sigma,\gamma) & \to q'(\tau,\beta)(\sigma,\gamma') & \text{for all } q \in Q,\ \sigma,\tau \in \Sigma \cup \{\varepsilon\},\ \gamma,\beta \in \Gamma, \\ & & \text{and } (q',\gamma',L) \in \delta(q,\gamma) \end{array}$$

We claim that the following holds:

Claim VI.5

$$(q_0, (2, \$\sigma_1\sigma_2\dots\sigma_n)) \vdash_M^* (q, (r, \$\gamma_1\gamma_2\dots\gamma_s))$$

iff

$$(\varepsilon, \$)q_0(\sigma_1, \sigma_1)\dots(\sigma_n, \sigma_n)(\varepsilon, \square)^{m-n} \Rightarrow_G^* (\varepsilon, \$)(\sigma_1, \gamma_1)(\sigma_2, \gamma_2)\dots(\sigma_{r-1}, \gamma_{r-1})q(\sigma_r, \gamma_r)\dots(\sigma_m, \gamma_m)$$

where $\sigma_i = \varepsilon$ if $i > n$, $\gamma_j = \square$ if $j > s$ and m is larger than the number of cells visited by M on $\sigma_1\sigma_2\dots\sigma_n$.

Proof sketch of the claim. The "$\Rightarrow$" direction is shown by induction on the length of the computation. For zero steps, the proof is trivial. Every production of the second part of the grammar G simulates one step of the Turing machine. So if we have a computation with n steps, we split it into a computation of $n-1$ steps and the last step. We apply the induction hypothesis to the computation with $n-1$ steps and then get a derivation by simulation the last step directly. The converse direction is done in the same manner, we just do induction on the length of the derivation. ∎

Note that if there is no production of the second part is applicable to a sentence, then this means that the Turing machine M stopped. If M accepts

$w = \sigma_1 \dots \sigma_n$, then G now has to produce w. It uses the first component of the symbols to achieve this.

$$\begin{aligned} q(\sigma,\gamma) &\mapsto q\sigma \quad \text{for all } q \in Q_{\text{acc}},\ \sigma \in \Sigma \cup \{\varepsilon\},\ \gamma \in \Gamma, \\ (\sigma,\gamma)q &\mapsto \sigma q \quad \text{for all } q \in Q_{\text{acc}},\ \sigma \in \Sigma \cup \{\varepsilon\},\ \gamma \in \Gamma, \\ q &\mapsto \varepsilon \quad \text{for all } q \in Q_{\text{acc}}. \end{aligned}$$

These productions remove the second track of the current sentence. Note how handy it is that every entry in the first component that is not a symbol of the input w is the empty word. In this way, these entries vanish when we concatenate. To get a terminal word, the state q can only be removed after all variables of the form (σ, γ) have been replaced by terminal symbols.

By construction, we have that M accepts a word w iff G produces w. This finishes the proof. ∎

Exercise VI.1 *Work out the details in the proof of the claim above.*

Altogether, we have shown the following result:

Theorem VI.6 *The class of all type-0 languages is precisely* RE.

VI.2 CSL versus linear bounded automata

A linear bounded automaton is a 1-tape nondeterministic Turing machine that cannot write on any other cell than the cells occupied by the input. The input w is bounded by \$ to the left and & to the right. When the Turing machine enters the \$, it has to write \$ and immediately to go to the right; if it enters & from the right, it has to go to the left.

It is quite easy to modify the proofs from the previous section to show that the languages recognized by linear bounded automata coincide with the set of context-sensitive languages.

Lemma VI.7 *Let $G = (V, \Sigma, P, S)$ be a type-1 grammar. Then there is a linear bounded automaton M such that $L(G) = L(M)$.*

Proof sketch. M works as the Turing machine in Lemma VI.3. Since it has only one tape, it uses a separate track for storing the current sentence. Since context-sensitive grammars are nondecreasing, M can stop whenever it wants to create a sentence that is longer than the input. ∎

Lemma VI.8 *Let M be a linear bounded automaton. Then there is a context-sensitive grammar G such that $L(M) = L(G)$.*

Proof sketch. First we modify the linear bounded automaton: Instead of an input $w = \sigma_1\sigma_2 \dots \sigma_{n-1}\sigma_n$, it gets the input $\bar{\sigma}_1\sigma_2 \dots \sigma_{n-1}\bar{\sigma}_n$. The left and the right most symbols are "marked" versions of the original symbols that allows the automaton to detect the begin and end of the input. Now it is easy to modify the automaton in such a way that it never leaves the input at all.

Again, we can modify the construction in Lemma VI.4. First we modify the first part of the grammar G in such a way that it can only generate sentences of the form $q_0(\sigma_1, \bar{\sigma}_1)(\sigma_2, \sigma_2) \dots (\sigma_{n-1}, \sigma_{n-1})(\sigma_n, \bar{\sigma}_n)$.

If we look at this new grammar, there is only one place at which we have productions whose righthand side is shorter than the lefthand side, namely the production $q \to \varepsilon$. We circumvent this by "packing" the state q into the neighbouring variables. More precisely, we replace occurrences of $q(\sigma, \gamma)$ by (σ, γ, q) and this is now one variable. It is easy to change the productions accordingly. In this way, the grammar G becomes nondecreasing. ∎

Altogether, we get the following theorem.

Theorem VI.9 (Kuroda) CSL *is precisely the class of languages recognized by linearly bounded automata.*

29 Context-free grammars

By definition, context-free grammars have at most one ε-production, namely, $S \to \varepsilon$, and if this production is present, then S is not allowed to appear on the righthand side of any production. It is often convenient to allow arbitrary ε-productions. For instance, the grammar

$$S \to \varepsilon \mid 0S1$$

looks much nicer than the one given by

$$\begin{aligned} S' &\to \varepsilon \mid S \\ S &\to 01 \mid 0S1 \end{aligned}$$

For context-free grammars (but not for context-sensitive grammars!), we can allow arbitrary ε-productions. So from now on, we allow arbitrary ε-productions in context-free grammars. We will see that we can always find an equivalent grammar that has at most one ε-production of the form $S \to \varepsilon$.

29.1 Derivation trees and ambiguity

Consider the grammar G given by

$$E \to E * E \mid E + E \mid (E) \mid x$$

It generates all arithmetic expressions with the operations $*$ and $+$ over the variable[1] x.[2] A word $w \in \Sigma^*$ is in $L(G)$ if $S \Rightarrow^* w$.[3] A derivation is a witness for the fact that $S \Rightarrow^* w$, i.e., a sequence of sentences such that $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_t \Rightarrow w$. Usually, a word w has many derivations. Here are two examples for Tthe word $x + x * x$ in the example above:

$$E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E * E \Rightarrow x + x * E \Rightarrow x + x * x \tag{29.1}$$

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * x \Rightarrow E + x * x \Rightarrow x + x * x \tag{29.2}$$

[1] This is a variable in the expression. Do not confuse this with a variable of the grammar.

[2] Yes, I know, arithmetic expressions over one variable without any constants are not really exciting. But you can replace x by your favourite collection of variables and constants.

[3] Whenever the grammar is clear from the context, we will write $S \Rightarrow^* w$ instead of $S \Rightarrow^*_G w$.[4]

[4] This could be a record number of footnotes per page.

In the first derivation, we always replace the leftmost variable. Such derivations are called *leftmost derivations*. In the second derivation, we always replace the rightmost variable. Such derivations are called, guess what, *rightmost derivations*. Although the two derivations are different, they are not "really different", they correspond to the same *derivation tree*.

Definition 29.1 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar.*

1. *A* derivation tree *(or* parse tree*) is an ordered tree with a node labeling such that:*
   (a) *The root is labeled with S.*
   (b) *All leaves are labeled with an element from $V \cup \Sigma$ or with ε. In the latter case, the leaf is the only child of its parent.*
   (c) *All interior nodes are labeled with an element of V. If A is this label and the labels of the children are $x_1, x_2, \ldots, x_t$ (in this order) then $A \to x_1 x_2 \ldots x_t \in P$.*[5]
2. *The* yield *(or* leaf-word *or* front *or ...)*[6] *of a parse tree is the concatenation of the labels of the leaves (in the order induced by the ordering of the vertices in the tree).*

Figure 29.1 shows the derivation tree that corresponds to the two derivations (29.1) and (29.2). The leftmost derivation (29.1) is obtained by doing a depth-first search and visiting the children from left to right, the rightmost derivation (29.2) is obtained by visiting them from right to left. In general, each derivation tree corresponds to exactly one left derivation and exactly one right derivation.

But there is another derivation tree for $x+x*x$. It is shown in Figure 29.2. Having several derivation trees for the same word is in general a bad thing.

Definition 29.2
1. *A context-free grammar is called* ambiguous *if there is a word that has two or more derivation trees. Otherwise the grammar is called* unambiguous.
2. *A context-free language is called* unambiguous *if there is an unambiguous grammar that generates the language. Otherwise it is called* inherently ambiguous.

The derivation tree in Figure 29.2 is unnatural, because it does not respect the usual precedence of the operators "$*$" and "$+$". But there is an

[5] If $t > 1$, then every $x_\tau \in V \cup \Sigma$. If $t = 1$, then $x_1 = \varepsilon$ is possible. In this case, $A \to \varepsilon$ is a production of P.

[6] There are far too many names for this.

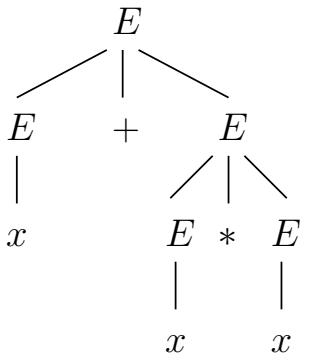


Figure 29.1: A derivation tree for $x + x * x$.

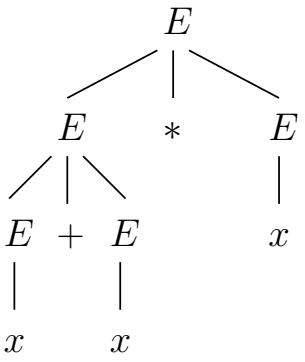


Figure 29.2: Another derivation tree for $x + x * x$.

unambiguous grammar:

$$E \to T \mid T + E$$
$$T \to F \mid F * T$$
$$F \to x \mid (E)$$

It is by no means obvious that the grammar above is unambiguous and this fact requires a formal proof. The proof is rather tedious and can be shown as follows:

- There is only one derivation tree for $T + T + \cdots + T$.
- There is only one derivation tree for deriving $F * F * \cdots * F$ from T. (This means that the root of the tree is labeled by T.) In turn, there is only one derivation tree for deriving $e_1 * e_2 * \cdots * e_t$ from T where each e_τ is either x or (E).
- Now we prove by the number of pairs of brackets that there is at most one derivation tree for every sentence: Take a pair of matching brackets in a sentence s. The subsentence s' between these brackets has to be derivable from E, since $F \to (E)$ is the only rule that introduces brackets and the chosen pair of brackets was a matching one. By the induction hypothesis, there is only one derivation tree for s'. If we replace the sentence (s') by F in s, then again by the induction hypothesis, there is only one derivation tree for this sentence.

Note that simply respecting the precedence of operators is not enough for making the grammar unambiguous: If we had taken the rule $T \to F \mid T * T$, then the grammar still would be ambiguous.

There are context-free languages that are inherently ambiguous.

Theorem 29.3 (without proof) *The language* $\{0^n 1^n 2^m 3^m \mid n, m \geq 1\} \cup \{0^n 1^m 2^m 3^n \mid n, m \geq 1\}$ *is context-free and inherently ambiguous.*

Exercise 29.1 *Show that the language from Theorem 29.3 is context-free.*

29.2 Elimination of useless variables

Assume you created a big context-free grammar for some, say, programming language. Can you find out whether everything is really needed in your grammar or are there some artifacts in it, i.e., variables that you do not need any more because you later changed some things somewhere else and now these variables do not occur in any derivation of a terminal word?

Definition 29.4 *Let* $G = (V, \Sigma, P, S)$ *be a context-free grammar.*

1. *A variable A is* generating *if there is a word $w \in \Sigma^*$ such that $A \Rightarrow_G^* w$.*
2. *A variable A is* reachable *if there are words $x, y \in (V \cup \Sigma)^*$ such that $S \Rightarrow_G^* xAy$.*
3. *A variable $A \in V$ is* useful *if it is reachable and generating. Otherwise, A is* useless.

Theorem 29.5 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar with $L(G) \neq \emptyset$. Let $H = (W, \Sigma, Q, S)$ be the grammar that is obtained as follows:*

1. *Remove all variables $A \in V$ that are not generating and all productions that contain A. Let the resulting grammar be $G' = (V', \Sigma, P', S)$.*
2. *Remove all variables $A \in V'$ that are not reachable in G' and all productions that contain A.*

Then $L(H) = L(G)$ and H does not contain any useless variables.

Proof. We start with some simple observation: If $S \Rightarrow^* uAv \Rightarrow^* w$ with $u, v \in (V \cup \Sigma)^*$ and $w \in \Sigma^*$, then A is both reachable and generating. The fact that A is reachable is obvious. But A is also generating, since we can derive some substring of w from it.

Next we show that H contains no useless variables:

Let $A \in W$. Since A survived the first step (as it is in W, it has to be in V'), there is a $w \in \Sigma^*$ such that $A \Rightarrow_G^* w$. Every variable in a corresponding derivation is also generating since we can derive a subword of w from it (cf. the observation above) and therefore, every variable in the derivation survives the first step and is in V'. Thus $A \Rightarrow_{G'}^* w$ and A is also generating in G'.

A variable $A \in W$ is certainly reachable in G' since it survived the second step. This means that there are $u, v \in (V' \cup \Sigma)^*$ such that $S \Rightarrow_{G'}^* uAv$. (Note that S is still in G', since $L(G) \neq \emptyset$.) But every variable in a correspoding derivation is also reachable in G'. Hence all variables in the derivation are in H and therefore, $S \Rightarrow_H^* uAv$. We know that every variable in the derivation is generating in G', thus $S \Rightarrow_H^* uAv \Rightarrow_{G'}^* w$ for some $w \in \Sigma^*$. But this means that every variable in a derivation corresponding to $uAv \Rightarrow_{G'}^* w$ is reachable from S in G'. Hence all of them are in H, too, and we have $S \Rightarrow_H^* uAv \Rightarrow_H^* w$. Thus A is generating and reachable, hence it is useful.

It remains to show that $L(G) = L(H)$:

"$\supseteq$": Is obvious since we only remove variables from G.

"$\subseteq$": If $S \Rightarrow_G^* w$ for some $w \in \Sigma^*$, then every variable in a derivation of w is both reachable and generating in G by the observation at the beginning of the proof. So all variables in the derivation survive the first step and are in G'. But since all variables are in G', we still have $S \Rightarrow_{G'}^* w$ and thus all variables in the derivation survive the second step. Therefore $S \Rightarrow_H^* w$ and $w \in L(H)$. ∎

Example 29.6 *Consider the following grammar*

$$S \to AB \mid 0$$
$$A \to 0$$

We cannot derive any terminal word from B, hence we remove the production $S \to AB$. Now we cannot derive any sentence of the form xAy from S, hence we remove the rule $A \to 0$, too. If we had reversed the order of the two steps, then we would not have removed anything in the first step and only the rule $S \to AB$ in the second step. The production $A \to 0$ would not have been removed.

Theorem 29.5 provides a way to eliminate useless symbols once we can determine the generating and reachable variables. Algorithms 12 and 13 solve these two tasks.

Exercise 29.2 *Show that the Algorithms 12 and 13 are indeed correct.*

Program 12 Determining the generating variables

Input: A context-free grammar $G = (V, \Sigma, P, S)$
Output: The set V_0 of all variables that are generating.

1: Add all $A \in V$ to V_0 for which there is a production $A \to u$ with $u \in \Sigma^*$.
2: **while** there is a production $A \to a_1 a_2 \dots a_t$ such that $A \notin V_0$ and all a_τ that are in V are in V_0 **do**
3: Add A to V_0.
4: **od**
5: Return V_0.

Program 13 Determining the reachable variables

Input: A context-free grammar $G = (V, \Sigma, P, S)$
Output: The set V_1 of all variables that are reachable

1: Add all $A \in V$ to V_1 for which there is a production $S \to xAy$ for some $x, y \in (V \cup \Sigma)^*$.
2: **while** there is a production $A \to a_1 a_2 \dots a_t$ such that $A \in V_1$ and some a_τ is in $V \setminus V_1$ **do**
3: Add all $a_\tau \in V \setminus V_1$ to V_1, $1 \leq \tau \leq t$.
4: **od**
5: Return V_1.

30 Chomsky normal form

In this chapter, we show a formal form for context-free grammars, the so-called *Chomsky normal form*. On the way, we also see how to eliminate ε-productions.

30.1 Elimination of ε-productions

Definition 30.1 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. $A \in V$ is called nullable if $A \Rightarrow_G^* \varepsilon$.*

Theorem 30.2 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Let $H = (V, \Sigma, Q, S)$ be the grammar that is generated as follows:*

1. *Replace every production $A \to a_1 a_2 \dots a_k$ by all 2^ℓ productions, one for each possibility to leave an a_{i_λ} out where $a_{i_1}, \dots, a_{i_\ell}$ are all nullable variables among $a_1, a_2, \dots, a_k$.*
2. *Remove every ε-production. (This removes also an ε-production that we might have introduced in the first step when $a_1, \dots, a_k$ are all nullable.)*

We have $L(G) \setminus \{\varepsilon\} = L(H)$.

Proof. We show the following:

For all $A \in V$ and $u \in (V \cup \Sigma)^*$: $A \Rightarrow_H^* u \iff A \Rightarrow_G^* u$ and $u \neq \varepsilon$.

From this statement, the claim of the theorem follows:
"$\Longrightarrow$": The proof is by induction on the length of the derivation:
Induction base: If $A \to u \in Q$, then $u \neq \varepsilon$ by construction. There is a production $A \to a_1 a_2 \dots a_t$, each $a_\tau \in V \cup \Sigma$, and indices $j_1, \dots, j_\ell$ such that the concatenation of all a_τ with $\tau \notin \{j_1, \dots, j_\ell\}$ is u and all a_τ with $\tau \in \{j_1, \dots, j_\ell\}$ are nullable. Thus $A \Rightarrow_G^* u$.
Induction step: If $A \Rightarrow_H^* u$, then $A \Rightarrow_H^* w \Rightarrow_H u$. This means that $w = xBz$ such that $B \to y \in Q$ and $u = xyz$. As in the proof of the induction base, we can show that $B \Rightarrow_G^* y$ holds. By the induction hypothesis, $A \Rightarrow_G^* xBz$. Altogether, $A \Rightarrow_G^* xyz = u$. $u \neq \varepsilon$, since $y \neq \varepsilon$.
"$\Longleftarrow$": Is left as an exercise. ∎

Exercise 30.1 *Show the "$\Longleftarrow$"-direction of the proof of Theorem 30.2.*

Theorem 30.2 provides a way to eliminate ε-productions once we can determine the nullable variables. Algorithm 14 solves the latter task.

Exercise 30.2 *Show that Algorithm 14 is indeed correct.*

Program 14 Determining the nullable variables

Input: A context-free grammar $G = (V, \Sigma, P, S)$
Output: The set V_0 of all variables that are nullable.

```
1: Add all A ∈ V to V_0 for which there is a production A → ε ∈ P.
2: while there is a production A → a_1 a_2 ... a_t such that A is not in V_0 and
   all a_τ are nullable do
3:    Add A to V_0.
4: od
5: Return V_0.
```

30.2 Elimination of chain productions

Definition 30.3 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. A production of the form $A \to B$ with $A, B \in V$ is called a* chain production.

Like ε-productions, chain productions are useful for getting compact grammars; $E \to T \mid E + T$ is an example. On the other hand, like ε-productions, chain productions are not desirable, because they do not generate anything really new. But again, there is a way to get rid of chain productions.

First of all, we can immediately remove all productions of the form $A \to A$. We build a directed graph $H = (V, E)$. There is an edge $(A, B) \in E$ if there is a chain rule $A \to B \in P$. (Recall that productions are tuples, therefore we can also write $E = P \cap (V \times V)$.) If H has a directed cycle, then there are productions $B_\tau \to B_{\tau+1} \in P$, $1 \leq \tau < t$, and $B_t \to B_1 \in P$. But this means that the variables $B_1, \ldots, B_t$ are interchangable. Whenever we have a sentence that contains a variable B_i we can replace this by any B_j by using the chain productions.

Exercise 30.3 *Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $H = (V, P \cap V \times V)$. Assume that there is a directed cycle in H consisting of nodes $B_1, B_2, \ldots, B_t$, $t \geq 2$. Let $G' = (V', \Sigma, P', S)$ be the grammar that we obtain by replacing all occurrences of a B_i by B_1, removing the variables $B_2, \ldots, B_t$ from V and the production of the form $B_1 \to B_1$. Show the following: $L(G) = L(G')$.*

If we apply the construction above several times, we obtain a grammar, call it again $G = (V, \Sigma, P, S)$, such that the corresponding graph $H = (V, P \cap V \times V)$ is acyclic.

Theorem 30.4 *Let $G = (V, \Sigma, P, S)$ be a context free grammar such that the graph $H = (V, P \cap V \times V)$ is acyclic. Then there is a grammar $G' = (V, \Sigma, P', S)$ without chain productions with $L(G) = L(G')$.*

Proof. The proof is by induction on the number of chain productions in P (or the number of edges in H).
Induction base: If there are no chain productions, then there is nothing to prove.
Induction step: Since H is acyclic and contains at least one edge, there must be one variable A that has indegree ≥ 1 but outdegree 0. Let $B_1, \ldots, B_t$ all variables such that $B_\tau \to A \in P$. Let $A \to u_1, \ldots, A \to u_\ell \in P$ all productions with lefthand side A. Since A has outdegree 0 in H, $u_\lambda \notin V$ for all $1 \leq \lambda \leq \ell$. We remove the productions $B_\tau \to A$ and replace them by $B_\tau \to u_\lambda$ for $1 \leq \tau \leq t$ and $1 \leq \lambda \leq \ell$. Let G'' be the resulting grammar. Since we removed at least one chain production and did not introduce any new ones, G'' has at least one chain production less than G. By the induction hypothesis, there is a grammar G' without any chain productions such that $L(G'') = L(G')$. Hence we are done if we can show that $L(G'') = L(G)$: If $S \Rightarrow_G^* w$ for some $w \in \Sigma^*$ and the production $B_\tau \to A$ is used in the corresponding derivation, then eventually, a production $A \to u_\lambda$ has to be used, too, since $w \in \Sigma^*$. Hence we can use the production $B_\tau \to u_\lambda$ directly and get a derivation in G'. Conversely, if $S \Rightarrow_{G'}^* w$ and the production $B_\tau \to u_\lambda$ is used in a corresponding derivation, then we can replace this step by two steps that use the productions $B_\tau \to A$ and $A \to u_\lambda$. ∎

30.3 The Chomsky normal form

Definition 30.5 *A context-free grammar $G = (V, \Sigma, P, S)$ is in* Chomsky normal form *if all its productions are either of the form*

$$A \to BC$$

or

$$A \to \sigma$$

with $A, B, C \in V$ and $\sigma \in \Sigma$.

Theorem 30.6 *For every context-free grammar $G = (V, \Sigma, P, S)$ with $L(G) \neq \emptyset$ there is a context-free grammar $G' = (V', \Sigma, P', S)$ in Chomsky normal form with $L(G') = L(G) \setminus \{\varepsilon\}$.*

Proof. By the result of the previous sections, we can assume that G does not contain any ε-productions and chain rules. Thereafter, $L(G)$ does not

contain the empty word anymore. For every $\sigma \in \Sigma$, we introduce a new variable T_σ, add the new production $T_\sigma \to \sigma$, and replace every occurence of σ in the productions in P by T_σ except in productions of the form $A \to \sigma$ (since this would introduce new chain productions, but $A \to \sigma$ has already the "right" form for Chomsky normal form).

Thereafter, every productions is either of the form $A \to \sigma$ or $A \to A_1 A_2 \dots A_t$ where $A_1, A_2, \dots, A_t$ are all variables and $t \geq 2$. Hence we are almost there except that the righthand sides might have too many variables. We can overcome this problem by introducing new variables $C_2, \dots, C_{t-1}$ and replacing the production $A \to A_1 A_2 \dots A_t$ by

$$\begin{aligned} A &\to A_1 C_2 \\ C_2 &\to A_2 C_3 \\ &\dots \\ C_{t-2} &\to A_{t-2} C_{t-1} \\ C_{t-1} &\to A_{t-1} A_t \end{aligned}$$

The resulting grammar G' is obviously in Chomsky normal form. It is easy to see that $L(G') = L(G) \setminus \{\varepsilon\}$ (see Exercise 30.4). ■

Exercise 30.4 *Prove that the grammar G' constructed in the proof of Theorem 30.6 indeed fulfills $L(G') = L(G) \setminus \{\varepsilon\}$ (and even $L(G') = L(G)$, since we assumed that G does not contain any ε-productions).*

Exercise 30.5 *Let G be a context-free grammar and let H be the grammar in Chomsky normal form constructed in this section. If G has p productions, how many productions can H have?*

30.4 Further exercises

A context-free grammar $G = (V, \Sigma, P, S)$ is in *Greibach normal form* if for every $A \to v \in P$, $v \in \Sigma V^*$.

Exercise 30.6 *Show that for every context-free grammar G, there is a context-free grammar H in Greibach normal form such that $L(G) \setminus \{\varepsilon\} = L(H)$. (Hint: First convert into Chomsky normal form.)*

31 The pumping lemma for context-free languages

In this chapter, we will develop a method for proving that a language is not context-free. It will be similar to the pumping lemma for regular languages.

31.1 The pumping lemma

Recall that a *binary* tree is a tree such that each node is either a leaf or has exactly two children. (Some people prefer to call a tree binary, if the number of children is at most two.) The *height* of a tree is the length of a longest path from the root to a leaf. The length of a path is the number of edges in it. (Some people prefer to take the number of nodes instead.) So a tree consisting of one single node has height 0.

Let G be a context-free grammar in Chomsky normal form. Derivation trees of a grammar in Chomsky normal form have a special structure: The parents of the leaves have only one child since they correspond to the productions of the form $A \to \sigma$. All other nodes have exactly two children since they correspond to the productions $A \to BC$. This means that if we remove the leaves from a derivation tree, we will get a binary tree.

Lemma 31.1 (Pumping lemma for context-free languages) *Let L be a context-free language. There is an $n \in \mathbb{N}$ such that for all words $w \in L$ with $|w| \geq n$, there are words u, x, y, z, v with $w = uxyzv$, $|xz| \geq 1$, $|xyz| \leq n$ and for all $i \in \mathbb{N}$, $ux^iyz^iv \in L$.*

Proof. Let $G = (V, \Sigma, P, S)$ be a grammar for $L \setminus \{\varepsilon\}$ in Chomsky normal form. Let $\nu = |V|$ and set $n = 2^\nu$. As mentioned above, after removing the leaves, a derivation tree T for w is always a binary tree. The number of leaves of this new tree T' is the same of the derivation tree, namely $|w| \geq 2^\nu$. By Lemma 31.2, T' has a path of length at least ν. On this path, there are $\nu + 1$ nodes. Thus at least two of these nodes are labeled with the same variable by the pigeonhole principle. Even stronger, we know that among the $\nu + 1$ nodes that are closest to the leaf on this path, two have the same label. Let this label be A. Figure 31.1 shows such a path.

Now we consider the two nested subtrees with the root labeled by A. These two subtrees determine the decomposition of w into $uxyzv$, see Figure 31.2. Since at the upper A, a production of the form $A \to BC$ is used, either $x \neq \varepsilon$ or $z \neq \varepsilon$, hence $|xz| \geq 1$. The height of the subtree with the

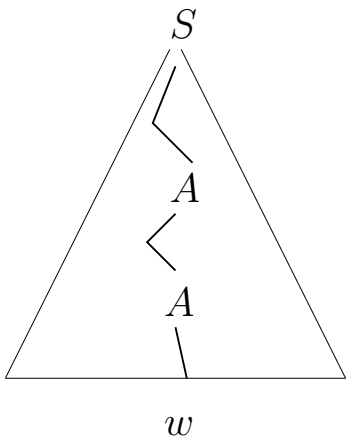


Figure 31.1: On a path of length ν, one variable, say A, has to appear twice.

upper A as root is at most ν by construction. Thus this subtree has a most $\leq 2^\nu = n$ leaves by Lemma 31.2. Thus $|xyz| \leq n$.

It remains to show that $ux^iyz^iv \in L$ for all $i \in \mathbb{N}$. We do this by constructing derivation trees for these words. We get a derivation tree for $ux^0yz^0v = uyv$ by removing the subtree with the upper A as root and replacing it by the subtree with the lower A as root. This is again a valid derivation tree, see Figure 31.3. We get a derivation tree for ux^2yz^2v by replacing the subtree with the lower A as root by a copy of the subtree with the upper A as root, see Figure 31.4. By repeating this replacement procedure, we get derivation trees for all ux^iyz^iv with $i \geq 2$. This completes the proof. ■

Lemma 31.2 *A binary tree of height h has at most $2^{h+1} - 1$ nodes and at most 2^h leaves.*

Proof. The proof is by induction on h.
Induction basis: A tree of height 0 has $1 = 2^{0+1} - 1$ nodes and $1 = 2^0$ leaves.
Induction step: Let T be a binary tree of height h. Let T_1 and T_2 be the subtrees adjacent to the root of T (see Figure 31.5). Let h_1 and h_2 be the heights of T_1 and T_2, respectively. We have $h_1, h_2 \leq h - 1$ (and for at least one of h_1 and h_2, equality holds). By the induction hypothesis, T_i has at most $2^{h_i+1} - 1$ nodes and 2^{h_i} leaves, $i = 1, 2$. Since $h_i \leq h - 1$, T has at most $2 \cdot (2^{h+1} - 1) + 1 = 2^{(h+1)+1} - 1$ nodes and $2 \cdot 2^h = 2^{h+1}$ leaves. ■

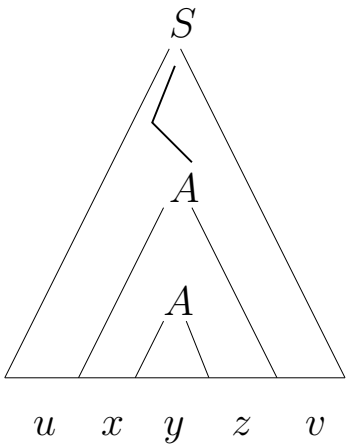


Figure 31.2: The subtrees with root A determine the decomposition of w into $uxyzv$.

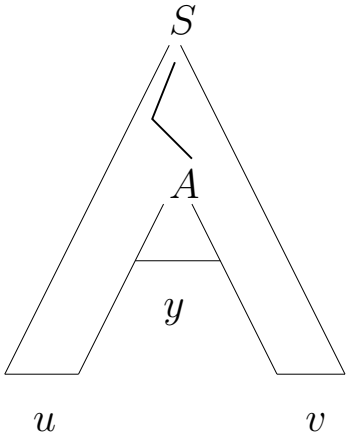


Figure 31.3: A derivation tree for ux^0yz^0v.

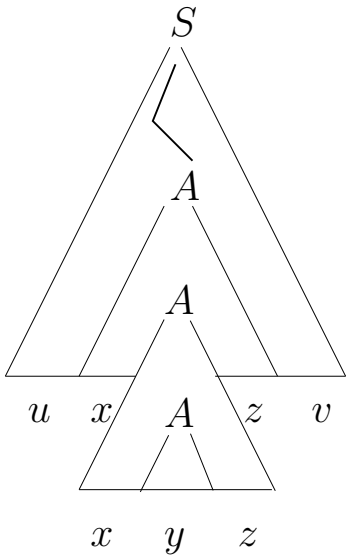


Figure 31.4: A derivation tree for ux^2yz^2v.

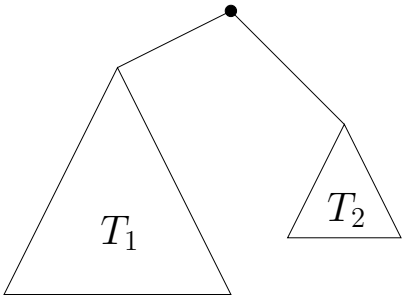


Figure 31.5: The tree T and its two subtrees T_1 and T_2.

31.2 How to apply the pumping lemma

Example 31.3 *Let us show that* $L = \{0^k1^k2^k \mid k \geq 1\}$ *is not context-free. Let* n *be an arbitrary natural number. We choose the word* $w = 0^n1^n2^n$*. Let* $w = uxyzv$ *be the decomposition as in the pumping lemma. Since* $|xyz| \leq n$*,* $xyz \in \{0,1\}^*$ *or* $xyz \in \{1,2\}^*$*. (In other words,* xyz *can never have* 0*'s and* 2*'s simultaneously, since the block of* 1*'s in the middle of* w *is too long.) If* L *were context-free, then* $ux^2yz^2v \in L$*. We show that* $ux^2yz^2v \notin L$ *and thus* L *cannot be context-free: We first assume that* xyz *consists of* 0*'s and* 1*'s. Since* $|xz| \geq 1$*, the number of* 0*'s or* 1*'s or both is greater than* n *but the number of* 2*'s does not change. Hence* $ux^2yz^2v \notin L$*. The case that* xyz *consists of* 1*'s and* 2*'s is treated similarly.*

Corollary 31.4 $\mathsf{CFL} \subsetneq \mathsf{CSL}$.

The pumping lemma game for context-free languages

Proving that a language L is not context-free via the pumping lemma can be considered as a game between you and your professor.

1. Your professor picks an $n \in \mathbb{N} \setminus \{0\}$.
2. You pick a word $w \in L$ with $|w| \geq n$.
3. Your professor picks words u, x, y, z, v such that $w = uxyzv$, $|xz| > 0$, and $|xyz| \leq n$.
4. You pick an $i \in \mathbb{N}$.

Now it comes to the showdown: You win if $ux^iyz^iv \notin L$. Your professor wins if $ux^iyz^iv \in L$. If you have a *winning strategy*, i.e, no matter what your professor picks, you can always make your choices such that you win, then L is not context-free.

(If L is indeed not context-free, this is one of the rare chances to win against your professor.)

The iron pumping lemma rule for context-free languages

The condition of the pumping lemma is only necessary.

Or in a more bureaucratic "rules language": The iron pumping lemma rule for regular languages applies accordingly.

31.3 How to decide properties of context-free languages

Which properties can we decide about context-free languages? We will discuss the same problems as for regular languages. The context-free languages will be given by a grammar G.

Word problem: Given G and $w \in \Sigma^*$, is $w \in L(G)$?

Emptiness problem: Given G, is $L(G) = \emptyset$?

Finiteness problem: Given G, is $|L(G)| < \infty$?

Equivalence problem: Given G_1 and G_2, is $L(G_1) = L(G_2)$?

31.3.1 The word problem

The question "$w \in L(G)$?" is decidable. We first compute a grammar $G' = (V', \Sigma, P', S)$ in Chomsky normal form for $L(G) \setminus \{\varepsilon\}$.[1] Then we systematically generate all derivations that generate sentences of length $\leq |w|$. The number of such derivations is bounded by $|P'|^{2|w|}$, since each application of a production either replaces a variable by a terminal or increases the length of the sentence by 1. After $2|w|$ such applications, we either generated w or a sentence from which we cannot derive w. We will see a much more efficient procedure in the exercises.

31.3.2 Testing emptiness

Again, we first compute a grammar $G' = (V', \Sigma, P, S)$ in Chomsky normal form for $L(G) \setminus \{\varepsilon\}$. $L(G')$ contains a word of length $< n := 2^{|V'|}$ iff $L(G') \neq \emptyset$. This follows from the pumping lemma. Assume we have a word $w \in L(G')$ with $|w| \geq n$. Then the pumping lemma says that there must be a shorter word in $L(G')$, namely the one that we get when we set $i = 0$. So we "just" have to test all words up to length n. But there is a much more efficient way: We just eliminate all useless variables. If some variables remain, then the language is nonemtpy.

31.3.3 Testing finiteness

Let G' and n be chosen as in Section 31.3.2. $L(G')$ is infinite iff $L(G')$ contains a word of length between n and $2n - 1$: If $L(G')$ is infinite, then it contains words that are arbitrarily long. By the pumping lemma, as long as the word has length $\geq n$, we can shorten it by an (unknown) amount between 1 and n. Thus we can bring it down to a length between n and $2n - 1$. On the other hand, if $L(G')$ contains a word of length $\geq n$, then it contains infinitely many, again by the pumping lemma. But again, there is a much more efficient way: We eliminate all useless variables and construct a graph $H = (V', E)$ where $(A, B) \in E$ iff there is a production of the form $A \to xBy$ with $x, y \in (V \cup \Sigma)^*$. $L(G')$ contains infinitely many words iff H contains a directed cycle.

31.3.4 Testing equivalence

Testing whether $L(G_1) = L(G_2)$ is an undecidable problem. The proof is by reduction to a problem called *Post's correspondence problem*. We will not present a proof in this lecture.

Exercise 31.1 *Show that the language of all pairs G_1 and G_2 of (encodings of) grammars with $L(G_1) \neq L(G_2)$ is in* RE.

[1] If $w = \varepsilon$, then we just can check whether S is nullable.

31.4 Further exercises

The pumping lemma for regular languages has one additional degree of freedom: We can choose the region of the word w where the pumping should take place. Here is a stronger version of the pumping lemma for the context-free languages that achieves something similar. It is usually called Ogden lemma, since it was proven by William Ogden in 1968. (And no, the pumping lemma was *not* proven by Mr. or Mrs. Pumping.)

The Ogden lemma speaks about *marked positions*. This simply means that we choose a bunch of symbols of w and these choices determine, to some extend, at which part of w the pumping takes place.

Lemma 31.5 (Ogden lemma) *Let L be a context-free language. There is an $n \in \mathbb{N}$ such that for all words $w \in L$ with $|w| \geq n$ and at least n positions marked, there are words u, x, y, z, v such that $w = uxyzv$, x or z have at least one marked position, x, y, and z together have at most n marked positions, and for all $i \in \mathbb{N}$, $ux^iyz^iv \in L$.*

Exercise 31.2 *Prove the Ogden lemma. (Hint: the proof is similar to the one of the pumping lemma. When constructing the path, nodes such that both subtrees contain marked positions play a crucial role.)*

Exercise 31.3 *Let* COPY $= \{ww \mid w \in \{0,1\}^*\}$. *Show that* COPY *is not context-free.*

Excursus: Are programming languages context-free?

We saw a context-free grammar for our programming language WHILE. In general, context-free languages describe the structure of computer programs quite well: It is easy to ensure that every **begin** has a corresponding **end**, that the structure of every arithmetic expression is correct, etc. But there is one thing that context-free grammars cannot describe. Consider the following simple program **int** x; $x := 0$. It declares x as an integer and then sets x to zero. Consider x as a place holder and replace it by an arbitrary string. In every typed programming language that I am aware of, the program above is considered to be correct whereas programs of the form **int** x; $y := 0$ with $x \neq y$ are not correct. But this is essentially the COPY language from Exercise 31.3. So context-free languages cannot ensure that every variable that is used is also declared.

Compilers of very complex languages, like C++, have to be really powerful. It can be shown that every C++ compiler necessarily is a universal WHILE program, see the reference below.

Martin Böhme, Bodo Manthey. The computational power of compiling C++. *Bulletin of the European Association for Theoretical Computer Science*, **81**:264-270, October 2003.

---

32 Pushdown automata

Type-3 languages are the regular languages, i.e., languages that are recognized by a (deterministic or nondeterministic) finite automaton. Are there (abstract) machines that characterize type-2 languages?

32.1 Formal definition

A *pushdown automaton* is a nondeterministic finite automaton that we equip with an additional stack. Alternatively, we can view a pushdown automaton as a restricted 2-tape Turing machine. One tape contains the input. This tape is oneway and read-only, that is, the Turing machine cannot move its head to the left and whenever it reads a symbol it has to write the same symbol. The second tape works like a stack: Whenever the Turing machine moves its head to the right, it has to erase the content of the current cell by a $\square$. There are no restrictions if the Turing machine moves its head to the left on the second tape.

Definition 32.1 *A* (nondeterministic) pushdown automaton *is described by a tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, Q_{\text{acc}})$ *where*

1. Q *is a finite set, the set of* states*,*
2. $\Sigma \subseteq \Gamma$ *is a finite set, the* input alphabet*,*
3. Γ *is a finite set, the* stack alphabet*,*
4. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_{\text{fin}}(Q \times \Gamma^*)$ *is the* transition function *(For a set* A*,* $\mathcal{P}_{\text{fin}}(A)$ *here denotes the set of all finite subsets of* A*),*
5. $q_0 \in Q$ *is the start symbol,*
6. $\# \in \Gamma \setminus \Sigma$ *is the bottom symbol of the stack, and*
7. Q_{acc} *is the set of accepting states.*

At the beginning of a computation, the input $w \in \Sigma^*$ is standing on the input tape and the head is on the first symbol of w. The only symbol in the stack is #.[1] The state of the automaton M is q_0.

[1] The pushdown automaton may write the symbol # on the stack if it wants but it does so at its own risk.

$(q', y) \in \delta(q, \sigma, \gamma)$ means that if M is in state q, reads σ on the input tape, and γ is the top symbol on the stack, then M can go to state q' and replace the top symbol γ by the symbols $y = y_1 y_2 \dots y_t$ where y_1 becomes the new top symbol. (This essentially means that we pop γ and then successively push $y_t, \dots, y_1$.) This operation can simulate a pop operation by choosing $y = \varepsilon$ and a push(τ) operation by setting $y = \tau\gamma$.

$(q', y) \in \delta(q, \varepsilon, \gamma)$ means that if M is in state q and γ is the top symbol of the stack, then M might go to state q' and replace γ by y as above without reading any symbol from the input ("ε-transition").

A *configuration* of M is a triple from $Q \times \Sigma^* \times \Gamma^*$. (q, v, z) means that M is in state q, v is the remainder of the input that M has not read so far, and z is the current content of the stack. The first symbol z_1 of z is supposed to be the top symbol of z.

Definition 32.2 *1. A* configuration *of a pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, Q_{\text{acc}})$ is a triple in $Q \times \Sigma^* \times \Gamma^*$.*

2. The start configuration on input $w \in \Sigma^*$ is $(q_0, w, \#)$.

3. We define a relation $\vdash_M$ on the set of configurations as follows:

$$(q, v_1 \dots v_n, z_1 \dots z_m) \vdash_M \begin{cases} (q', v_2 \dots v_n, y_1 \dots y_t z_2 \dots z_m) & \text{if } (q', y_1 \dots y_t) \in \delta(q, v_1, z_1) \\ (q', v_1 \dots v_n, y_1 \dots y_t z_2 \dots z_m) & \text{if } (q', y_1 \dots y_t) \in \delta(q, \varepsilon, z_1) \end{cases}$$

Above, $m \geq 1$ and in the first line of the case distinction, $n \geq 1$.

4. $\vdash_M^*$ denotes the transitive and reflexive closure of $\vdash_M$.

Definition 32.3 *1. A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, Q_{\text{acc}})$* accepts *a word w* by empty stack *if $(q_0, w, \#) \vdash_M^* (q, \varepsilon, \varepsilon)$ for some $q \in Q$.*

2. $N(M) = \{w \in \Sigma^* \mid M \text{ accepts } w \text{ by empty stack}\}$.

3. M accepts *w* by accepting state *if $(q_0, w, \#) \vdash_M^* (q, \varepsilon, z)$ for some $q \in Q_{\text{acc}}$ and $z \in \Gamma^*$.*

4. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w \text{ by accepting state}\}$.

Note that in general $L(M) \neq N(M)$. But we will see that for every pushdown automaton M, there is another pushdown automaton M' such that $L(M) = N(M')$ and vice versa.

Exercise 32.1 *Show that if we modify the transition function to $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to Q \times (\Gamma \cup \{\varepsilon\})$ where $(q', \gamma) \in \delta(q, \sigma, \varepsilon)$ means that we append γ to the stack without reading the top symbol of the stack, then we still can simulate pushdown automata as above.*

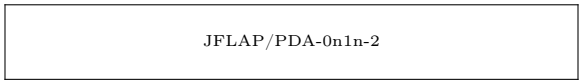


Figure 32.1: The pushdown automaton for $\{xx^{\text{rev}} \mid x \in \{0,1\}^*\}$.

As we did for nondeterministic finite automata, we can define the *computation tree* of M on w. The root of the tree is labeled with the start configuration $(q_0, w, \#)$. Whenever we have a node that is labeled with a configuration c and $c_1, \ldots, c_t$ are the configurations with $c \vdash_M c_\tau$, $1 \leq \tau \leq t$, then c gets t children, each labeled with $c_1, \ldots, c_t$. There might be a node in the tree that is already labeled with c_τ but we nevertheless always add a new node.

Example 32.4 *Figure 32.1 shows a pushdown automaton M with $L(M) = \{xx^{\text{rev}} \mid x \in \{0,1\}^*\}$. (Since the automaton was constructed with the JFLAP tool, the bottom symbol of the stack is denoted by Z.) It has three states, push, pop, and end. In the state push, it pushes the symbol read on the stack. Then it can nondeterministically go to the state pop. We say that M nondeterministically* guesses *the middle of the word. In the state pop, M pops the symbols from the stack and compares them with the remaining symbols of the input. If there is a mismatch, there is no possible transition and the computation stops. If the automaton reaches the Z, then it pops the Z without reading the input symbol and goes to the accepting state end. If M read the whole input, then it accepts.*

Why do finite automata have no configurations?

Well, they have. A configuration consists of the state and the part of the input not read so far. But the δ^* function provides the same information as the $\vdash$ relation does in the corresponding reachability graph. So there is no need to define configurations for finite automata.

32.2 Empty stack versus accepting state

Lemma 32.5 *For every pushdown automaton M there is a pushdown automaton M' with $L(M) = N(M')$.*

Proof overview: M' simulates M step by step. When M enters an accepting state, then M' can enter a new state s by an ε-transition. In s, M pops all the symbols from the stack until it is empty.

There is only one problem: In a nonaccepting computation of M, M might empty the stack. This does not matter for M, but if now M' empties the stack in the simulation it might accept an input that M does not accept. To avoid this, M' replaces # by #\$ in the beginning, i.e., it places a new symbol \$ at the bottom of the stack. This symbol can only be removed in the state s. In this way, M' never empties the stack while simulating M.

Exercise 32.2 *Prove Lemma 32.5 formally.*

Lemma 32.6 *For every pushdown automaton M there is a pushdown automaton M' with $N(M) = L(M')$.*

Proof overview: M' simulates M step by step. Whenever M' empties the stack, M should enter a new accepting state by an ε-transition. However, when M' empties the stack, then the computation stops since δ is only defined if there is a symbol on the stack. To avoid this, we again place a new symbol at the bottom of the stack.

Exercise 32.3 *Prove Lemma 32.6 formally.*

33 Pushdown automata versus context-free grammars

33.1 Pushdown automata can simulate context-free grammars

In this section, we show that nondeterministic pushdown automata recognize the context-free languages. We start with a simple observation:

Observation 33.1 *Let M be some pushdown automaton and let (q, w, yz) be some configuration where $y, z \in \Gamma^*$. If $(q, w, yz) \vdash^* (p, v, z)$ and no symbol of z is ever removed from the stack during the corresponding computation, then $(q, w, y) \vdash^* (p, v, \varepsilon)$.*

Theorem 33.2 *For every context-free grammar $G = (V, \Sigma, P, S)$, there is a pushdown automaton M such that $L(G) = N(M)$.*

Proof overview: The automaton tries to construct a left derivation for a given word w. M starts with S on the stack. If the top symbol of the stack is a variable A, then M nondeterministically chooses some $A \to y \in P$ and expands A by replacing it by y. If the top symbol of the stack is a terminal, then we know that this terminal will always stay in the first position since it is not preceeded by any variable. Thus this symbol has to match the currently first symbol of the input. We read this symbol and compare it with the top symbol of the stack. If they are the same, then we pop the top symbol and go on. If they differ, then M stops (and does not accept).

Proof. We set $M = (\{q\}, \Sigma, V \cup \Sigma, \delta, q, S, \emptyset)$. M has only one state. Since we accept by empty stack, we do not need any accepting states. The transition function δ is defined as follows

$$\begin{array}{lcll} \delta(q, \varepsilon, A) & = & \{(q, y) \mid A \to y \in P\} & \text{for all } A \in V \\ \delta(q, \sigma, \sigma) & = & \{(q, \varepsilon)\} & \text{for all } \sigma \in \Sigma. \end{array}$$

Now we have to show: For all $w \in \Sigma^*$, $w \in N(M) \iff w \in L(G)$

"$\Longleftarrow$": Let $S = y_1 \Rightarrow_G y_2 \Rightarrow_G \ldots \Rightarrow_G y_n = w$ be a left derivation. We claim that

$$(q, w, S) \vdash_M^* (q, v_i, z_i) \quad \text{for all } 1 \leq i \leq n$$

where v_i and z_i are defined by

$$y_i = u_i z_i, \quad u_i \text{ is the longest prefix of } y_i \text{ with } u_i \in \Sigma^*$$
$$w = u_i v_i.$$

The proof of the claim is by induction on i.
Induction basis: We have $y_1 = S$. Hence $u_1 = \varepsilon$ and $v_1 = w$. $(q, w, S) \vdash^* (q, w, S)$ certainly holds.
Induction step: By the induction hypothesis, $(q, w, S) \vdash^* (q, v_i, z_i)$. We have to show that $(q, w, S) \vdash^* (q, v_{i+1}, z_{i+1})$. For this, it is sufficient to show that $(q, v_i, z_i) \vdash^* (q, v_{i+1}, z_{i+1})$.

By construction of u_i, z_i starts with a variable, say A. The step $y_i \Rightarrow y_{i+1}$ replaces the variable A by some righthand side of a production in P, say $A \to x$. We have $(q, x) \in \delta(q, \varepsilon, A)$ by construction, hence M can replace A on the stack by x. With the transitions $(q, \varepsilon) \in \delta(q, \sigma, \sigma)$, M can now pop all the terminals on top of the stack until it reaches the first variable. (Note that all input symbols must match the symbols on the stack since we are mimicking a valid derivation.) Hence $(q, v_i, z_i) \vdash^* (q, v_{i+1}, z_{i+1})$. This finishes the proof of the claim.

From the claim, the "$\Longleftarrow$"-direction immediately follows by noting that $v_n = \varepsilon$, since $y_n = w$ and $z_n = \varepsilon$.
"$\Longrightarrow$": We claim that for all $x \in \Sigma^*$ and $A \in V$

$$(q, x, A) \vdash_M^* (q, \varepsilon, \varepsilon) \quad \text{implies} \quad A \Rightarrow_G^* x.$$

The proof is by induction in n, the number of steps that M makes.
Induction basis: If $n = 1$, then $A \to \varepsilon \in P$ and $x = \varepsilon$. But then $A \Rightarrow^* x$.
Induction step: Since the top symbol of the stack is a variable, the first step of M replaces A by some righthand side of some production. Let $A \to a_1 \dots a_k$ be the chosen production with $a_\kappa \in V \cup \Sigma$, $1 \le \kappa \le k$. In the next $n-1$ steps, M consumes the whole input and eventually removes the symbol $a_1, \dots, a_k$ from the stack. Decompose $x_1 = y_1 y_2 \dots y_k$ with $y_i \in \Sigma^*$, $1 \le i \le k$, such that $y_1 y_2 \dots y_\kappa$ are the symbols read when a_κ is removed from the stack. (Here we mean the occurrence of a_κ that is put on the stack when we replace A in the beginning.)

By Observation 33.1,

$$(q, y_\kappa y_{\kappa+1} \dots y_k, a_\kappa) \vdash^* (q, y_{\kappa+1} \dots y_k, \varepsilon), \quad 1 \le \kappa \le k.$$

This implies

$$(q, y_\kappa, a_\kappa) \vdash^* (q, \varepsilon, \varepsilon) \quad 1 \le \kappa \le k.$$

By the induction hypothesis, $a_\kappa \Rightarrow^* y_\kappa$ if $a_\kappa \in V$. If $a_\kappa \in \Sigma$, then $a_\kappa = y_\kappa$ and $a_\kappa \Rightarrow y_\kappa$ trivally holds. Altogether

$$A \Rightarrow a_1 \dots a_k \Rightarrow^* y_1 \dots y_k = x.$$

The "$\Longrightarrow$"-direction now follows from the claim by setting $A = S$ and $x = w$. ∎

33.2 Context-free grammars can simulate pushdown automata

Theorem 33.3 *For every pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, \emptyset)$, there is a context-free grammar G with $N(M) = L(G)$*

Proof overview: G should generate a word w iff M started in $(q_0, w, \#)$ reads w completely and empties its stack, i.e, $(q_0, w, \#) \vdash_M^* (p, \varepsilon, \varepsilon)$ for some $p \in Q$. We construct G by solving a more general problem. For every $q, p \in Q$ and $\gamma \in \Gamma$, G has a variable $[q, \gamma, p]$ such that $[q, \gamma, p] \Rightarrow_G^* w$ iff M started in (q, w, γ), reads w completely and empties its stack, i.e, $(q, w, \gamma) \vdash_M^* (p, \varepsilon, \varepsilon)$.

Then $w \in N(M)$ iff $[q_0, \#, p] \Rightarrow^* w$ for some $p \in Q$. The reason why we solve the more general problem is that the original problem is split into several instances of the more general problem, which we solve recursively.

Proof. We define $G = (V, \Sigma, P, S)$ as follows:

- $V = \{S\} \cup \{[q, \gamma, p] \mid q, p \in Q, \gamma \in \Gamma\}$.
- There are "starting rules"

  $$S \to [q_0, \#, p]$$

  for every $p \in Q$.
- And there are "transition rules"

  $$[q, \gamma, r_k] \to \sigma[r, y_1, r_1][r_1, y_2, r_2] \dots [r_{k-1}, y_k, r_k]$$

  for every $\sigma \in \Sigma \cup \{\varepsilon\}$, $\gamma \in \Gamma$, and for every $(r, y_1 y_2 \dots y_k) \in \delta(q, \sigma, \gamma)$ and for every possible sequence of states $r_1, \dots, r_k \in Q$. (If $k = 0$, then the production is just $[q, \gamma, r] \to \sigma$.)

We claim that

$$[q, \gamma, p] \Rightarrow^* w \iff (q, w, \gamma) \vdash^* (p, \varepsilon, \varepsilon).$$

for all $q, p \in Q$, $\gamma \in \Gamma$, and $w \in \Sigma^*$.

"$\Longleftarrow$": The proof is by induction on n, the number of steps of M.
Induction basis: If $n = 1$, then $(p, \varepsilon) \in \delta(q, w, \gamma)$ and $w \in \Sigma \cup \{\varepsilon\}$. By construction, $[q, \gamma, p] \to w \in P$.
Induction step: Assume that $(q, w, \gamma) \vdash^* (p, \varepsilon, \varepsilon)$ is a computation of n steps. Let $(q, w, \gamma) \vdash (r_0, v, y_1 \dots y_k)$ be the first step. Then $w = \sigma v$ for some $\sigma \in \Sigma \cup \{\varepsilon\}$ and $(r_0, y_1 \dots y_k) \in \delta(q, \sigma, \gamma)$.

By construction, there is a production

$$[q, \gamma, r_k] \to \sigma[r_0, y_1, r_1][r_1, y_2, r_2] \dots [r_{k-1}, y_k, r_k]$$

Decompose $v = u_1 \dots u_k$ with $u_\kappa \in \Sigma^*$, $1 \leq \kappa \leq k$ in such a way that $u_1 \dots u_\kappa$ is the input read when y_κ is removed from the stack. (We here mean the occurrence of y_κ that is put on the stack in the first step.) By Observation 33.1 above,

$$(r_{\kappa-1}, u_\kappa, y_\kappa) \vdash^* (r_\kappa, \varepsilon, \varepsilon), \qquad 1 \leq \kappa \leq k.$$

Now we can apply the induction hypothesis, since the computations are shorter than n. Thus we get

$$[r_{\kappa-1}, y_\kappa, r_\kappa] \Rightarrow^* u_\kappa, \qquad 1 \leq \kappa \leq k.$$

Altogether, we get

$$[q, \gamma, r_k] \Rightarrow \sigma[r, y_1, r_1] \dots [r_{k-1}, y_k, r_k] \Rightarrow \sigma u_1 \dots u_k = w.$$

"$\Longrightarrow$": Is left as an exercise.

Now the theorem follows from the claim, since

$$\begin{aligned} S \Rightarrow^* w &\iff [q_0, \#, p] \Rightarrow w \quad \text{for some } p \in Q \\ &\iff (q_0, w, \#) \vdash^* (p, \varepsilon, \varepsilon). \end{aligned}$$ ∎

Exercise 33.1 *Prove the $\Longrightarrow$-direction of the proof of the claim in Theorem 33.3.*

Bronze rule of nondeterminism

Always remember the silver rule of nondeterminism

Nondeterministic Pushdown automata are interesting, because they precisely characterize the context-free languages, not because we can build them in an efficient way. Interesting for compiler construction etc. are *deterministic* pushdown automata. But there are context-free languages that are not recognized by a deterministic pushdown automaton.

Type-2 languages (context-free languages)

Let $L \subseteq \Sigma^*$. The following statements are equivalent:

- There is a context-free grammar (type-2 grammar) G with $L = L(G)$.
- There is a context-free grammar G with ε-productions with $L = L(G)$.
- There is a context-free grammar G in Chomsky normal form with $L \setminus \{\varepsilon\} = L(G)$.
- There is a nondeterministic pushdown automaton M with $L(M) = L$.
- There is a nondeterministic pushdown automaton M with $N(M) = L$.

33.3 Deterministic context-free languages

Definition 33.4 *A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, Q_{\text{acc}})$ is* deterministic *if*

1. $|\delta(q, \sigma, \gamma)| \leq 1$ *for all* $q \in Q$, $\sigma \in \Sigma \cup \{\varepsilon\}$, *and* $\gamma \in \Gamma$.
2. *If* $|\delta(q, \sigma, \gamma)| = 1$ *for some* $\sigma \in \Sigma$, *then* $\delta(q, \varepsilon, \gamma) = \emptyset$.

Given the state, the current symbol of the input, and the top symbol of the stack, a deterministic pushdown automaton has at most one choice. If M is in state q and the top symbol of the stack is γ and M has the choice to make an ε-transition, then no other transition is possible, because otherwise we could simulate any nondeterministic pushdown automaton.

The advantage of deterministic pushdown automata is that the word problem (given w and M, is $w \in L(M)$?) is decidable in linear time (by the automaton itself!). M might run into an infinite loop but such loops can be easily detected (and removed) in advance.

For deterministic pushdown automata it makes a difference whether they accept by empty stack or by accepting state.

Theorem 33.5 (without proof) *Let $L \subseteq \Sigma^*$. Then the following two statements are equivalent:*

1. $L = N(M)$ *for some deterministic pushdown automaton M.*

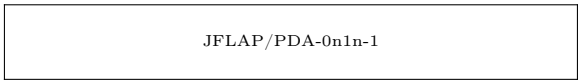


Figure 33.1: The pushdown automaton for $\{x\$x^{\text{rev}} \mid x \in \{0,1\}^*\}$.

2. *$L = L(M)$ for some deterministic pushdown automaton M and no $x \in L$ is a proper prefix of some $y \in L$*

If we want to accept by empty stack and do not want to lose any computational power compared to accepting by accepting state, we have to mark the end of the input, that is, instead of L, we construct an automaton for $L\{\$\}$ where \$ is some new symbol. This new language $L\{\$\}$ has the property that no $x \in L\{\$\}$ is a proper prefix of a $y \in L\{\$\}$.

Definition 33.6

$$\mathsf{DCFL} = \{L \mid L = L(M) \text{ for some deterministic pushdown automaton } M\}.$$

There are context-free languages that are not in DCFL. $L = \{xx^{\text{rev}} \mid x \in \{0,1\}^*\}$ is such an example. The proof that $L \notin \mathsf{DCFL}$ is rather elaborate. We will not present it here. The idea behind this is the following: A deterministic pushdown automaton M for L has to accept 0^n110^n. To check whether 0^n110^m is in L, M pushes the first n zeros on the stack and then pops a zero for every 0 it reads after the 11. After this it has lost all information about n. M also has to accept $0^n110^n0^n110^n$. But it cannot distinguish this from $0^n110^n0^m110^m$, since after reading 0^n110^n it lost all information about n. (This is *not* a formal proof! Never write something like this in an exam.)

On the other hand, there are deterministic context-free languages that are not regular. $L' = \{x\$x^{\text{rev}} \mid x \in \{0,1\}^*\}$ is an example. The difference to L above is that we here tell the automaton the middle of the word. Figure 33.1 shows a deterministic finite automaton for L'.

Theorem 33.7 $\mathsf{REG} \subsetneq \mathsf{DCFL} \subsetneq \mathsf{CFL}$.

33.4 Further exercises

Exercise 33.2 *Show that a pushdown automaton with two stacks can simulate a 1-tape Turing machine.*

Bibliography

[BHPS61] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. On formal properties of simple phrase, structure grammars. *Z. Phonetik, Sprachwiss. Kommun.*, 14:143–172, 1961.

[Brz63] Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Proc. Sympos. Math. Theory of Automata (New York, 1962)*, pages 529–561. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., 1963.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Trans. Information Theory*, 2(3):113–124, 1956.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HS03] Juraj Hromkovic and Georg Schnitger. Nondeterminism versus determinism for two-way finite automata: Generalizations of Sipser's separation. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 439–451. Springer, 2003.

[Jaf78] Jeffrey Jaffe. A necessary and sufficient pumping lemma for regular languages. *ACM SIGACT News*, 10:48–49, 1978.

[Kle43] Stephen Cole Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41—-73, 1943.

[Myh57] J. Myhill. Finite automata and representation of events. Technical note WADC 57-624, Wright Patterson AFB, Dayton, Ohio, 1957.

[Ner58] Anil Nerode. Linear automaton transformations. *Proc. Amer. Math. Soc.*, 9:541–544, 1958.

[SM76] Joel I. Seiferas and Robert McNaughton. Regularity-preserving relations. *Theor. Comput. Sci.*, 2(2):147–154, 1976.

[SW82] Donald F. Stanat and Stephen F. Weiss. A pumping theorem for regular languages. *ACM SIGACT News*, 14:36–37, 1982.