Part I

Computability
0 Prelude: Sets, relations, words

0.1 Sets

A set is a “collection of objects”. While this is a rather naïve definition that has some pitfalls, it is sufficient for our needs here. Axiomatic set theory is beyond the scope of this lecture. A set \( A \) that contains the numbers 2, 3, and 5 is denoted by \( A = \{2, 3, 5\} \). 2, 3, and 5 are the elements of \( A \). We write \( 2 \in A \) to indicate that 2 is an element of \( A \) and \( 4 \notin A \) to indicate that 4 is not. If a set \( A \) contains only a finite number of elements, then we call it finite and the number of elements is the size of \( A \), which is denoted by \( |A| \).

Infinite sets cannot be denoted like this, we denote them for instance as \( \{0, 1, 2, 3, \ldots\} \) and hope for your intuition to fill in the remaining numbers. The last set, as you already guessed, is the set of natural numbers. We will also use the symbol \( \mathbb{N} \) for it. Note that \( 0 \in \mathbb{N} \).

A set \( B \) is a subset of \( A \) if every element of \( B \) is also an element of \( A \). In this case, we write \( B \subseteq A \). There is one distinguished set that contains no elements, the empty set; we denote it by \( \emptyset \). The empty set is a subset of every other set.

There are various operations on sets. The union \( A \cup B \) of two sets \( A \) and \( B \) is the set that contains all elements that are in \( A \) or in \( B \). The intersection \( A \cap B \) is the set that contains all elements that are simultaneously in \( A \) and in \( B \). For instance, the union of \( \{2, 3, 5\} \) and \( \{2, 4\} \) is \( \{2, 3, 4, 5\} \), their intersection is \( \{2\} \). The cartesian product \( A \times B \) is the set of ordered tuples \( \{(a, b) \mid a \in A, b \in B\} \). The cartesian product of \( \{2, 3, 5\} \) and \( \{2, 4\} \) consists of the six tuples \( \{(2, 2), (2, 4), (3, 2), (3, 4), (5, 2), (5, 4)\} \).

0.2 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 0.1 \( R_1 = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\} \subseteq \{1, 2, 3\} \times \{1, 2, 3\} \) 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.

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, $R_1$ 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.)

**Exercise 0.1** 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

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

Another important type of relations are equivalence relations. These are relations that are reflexive, symmetric and transitive. $R_1$ is not an equivalence relation, it is not symmetric. Neither is $R_2$ since it is not reflexive, among other things.

**Example 0.2** $R_3 = \{(1, 1), (2, 2), (3, 3)\}$ is the equality relation on $\{1, 2, 3\}$.

$R_3$ is an equivalence relation. It is easy to check that it is reflexive and it is trivially transitive and symmetric since there are no $(a, b) \in R_3$ with $a \neq b$. If $R$ is an equivalence relation, 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.
Exercise 0.2 Proof the following: The equivalence classes of an equivalence relation $R$ on $A$ are a partition of $A$, i.e., every $a \in A$ belongs to exactly one equivalence class of $R$.

0.3 Functions

A relation $f$ on $A \times B$ is called a partial function if for all $a \in A$ there is at most one $b \in B$ such that $(a, b) \in f$. If such a $b$ exists, we also write $f(a) = b$. If such a $b$ does not exist, we say that $f(a)$ is undefined. Occasionally, we write $f(a) = \text{undefined}$, although this not quite correct, since it suggests that “undefined” is an element of $B$. $\text{dom } f$ denotes the set of all $a$ such that $f(a)$ is defined. $f$ is called total if $\text{dom } f = A$.

A total function $f$ is called injective if for all $a, a' \in A$, $f(a) = f(a')$ implies $a = a'$. The image $\text{im } f$ of a function is the set $\{f(a) \mid a \in A\}$. $f$ is called surjective if $\text{im } f = B$.

A total function $f$ is called bijective if it is injective and surjective. In this case, for every $a \in A$ there is exactly one $b \in B$ such that $f(a) = b$. In this case, we can define the inverse $f^{-1}$ of $f$, which is a function $B \to A$: $f^{-1}(b) = a$ if $f(a) = b$.

0.4 Words

Let $\Sigma$ be a finite nonempty set. In the context of words, $\Sigma$ is usually called an alphabet. The elements of $\Sigma$ are called symbols or letters. A (finite) word $w$ over $\Sigma$ is a finite sequence of elements from $\Sigma$, i.e., it is a function $w : \{1, \ldots, \ell\} \to \Sigma$ for some $\ell \in \mathbb{N}$. $\ell$ 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 $\varepsilon$. Formally it is the sequence $\emptyset \to \Sigma$.

Example 0.3 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 we can write $w$ in an even more compact form as $w_1w_2 \ldots w_n$. $w_i$ is also called the $i$th symbol of $w$. The word $u$ from the example above can be written as $aba$.

The set of all words of length $n$ over $\Sigma$ is denoted by $\Sigma^n$. This usually denotes the set of all $n$-tuples with entries from $\Sigma$, 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 $\Sigma^*$.

One important operation is concatenation. Informally, it is the word that we get when juxtaposing two words. Formally it is defined as follows.
Let $w : \{1, \ldots, \ell\} \to \Sigma$ and $x : \{1, \ldots, k\} \to \Sigma$ be two words. Then the concatenation of $w$ and $x$ is the function
\[
\{1, \ldots, \ell + k\} \to \Sigma
\]
\[
i \mapsto \begin{cases} 
w(i) & \text{if } 1 \leq i \leq \ell \\
x(i - \ell) & \text{if } \ell + 1 \leq i \leq \ell + k.
\end{cases}
\]
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 $w$.

**Exercise 0.3** Let $\Sigma$ be an alphabet. Prove the following:

1. The relation “the length of $a$ is smaller than or equal to the length of $b$” is a partial order on $\Sigma^*$. If $|\Sigma| = 1$, then the order is even total.

2. The relation “the length of $a$ equals the length of $b$” is an equivalence relation.

**Exercise 0.4** Let $\Sigma$ be an alphabet and let $R$ be a total order on $\Sigma$. The lexicographic order $\leq_{\text{lex}}^R$ with respect to $R$ on $\Sigma^n$ is defined as follows:
\[
u_1 \nu_2 \ldots \nu_n \leq_{\text{lex}}^R \nu_1 \nu_2 \ldots \nu_n \text{ if } u_i R v_i \text{ where } i = \min\{1 \leq j \leq n \mid u_j \neq v_j\}
\]
or $i$ does not exist.

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

During the last year you learnt what a computer can do: how to model a problem, how to develop an algorithm, how to implement it and how to test your program. In this lecture, you will learn what a computer cannot do. 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. 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, 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

$^1$This is just at an intuitive level. We will formalize this soon.

$^2$Now it is—especially since you are studying at Saarland University—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

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.


Exercise 1.1 Goldbach’s conjecture states that every even number $\geq 4$ 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.
2 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 typically not 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.

2.1 Syntax

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

- variables: $x_0, x_1, x_2, \ldots$
- constants: $0, 1, 2, \ldots$
- 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. Please do not send any complaint emails like last year.)

Definition 2.1 WHILE programs are defined inductively:

1. A simple statement is of the form

   $x_i := x_j + x_k$ or $x_i := x_j - x_k$ 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:

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

Exercise 2.1 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 2.1 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 2.2** FOR programs are defined inductively:

1. A simple statement is of the form

\[
x_i := x_j + x_k \quad \text{or} \quad x_i := x_j - x_k \quad \text{or} \quad x_i := c,
\]

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) \text{for } x_i \text{ do } P_1 \text{ od} \quad \text{or} \quad (b) \text{ } 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.

### 2.2 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. The set \( X = \{x_0, x_1, x_2, \ldots \} \) 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 \( \ell \geq s - 1 \). A state is a vector \( S \in \mathbb{N}^{\ell+1} \). It describes the
content of the variables in $P$. If $\alpha_0, \ldots, \alpha_{s-1}$ is our input, then the initial state $S_0$ will be $(\alpha_0, \ldots, \alpha_{s-1}, 0, \ldots, 0)$.

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 $P$. But it becomes a state of $P$ if we append two more entries to the vector, filled with 0. 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 $n \geq \ell_0$. Since only the values of $S$ up to $\ell_0$ are interesting, we will just write down the values up to $\ell_0$ and often, we treat $S$ as a finite vector in $\mathbb{N}^{\ell_0+1}$.

Given a state $S$ and a program $P \in W_n$, 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 2.1 (or Definition 2.2), i.e., these programs are in $W_{n-1}$.

We now define a function $\Phi$ that defines the semantics of WHILE/FOR programs. Let $\Phi_P(S)$ denote the state that is reached after running $P$ on state $S = (\sigma_0, \ldots, \sigma_{\ell})$. $\Phi_P$ will be a partial function, i.e., $\Phi_P(S)$ might be undefined (in the case where $P$ does not halt on $S$). WHILE programs may not halt; at an intuitive level—we did not define the semantics so far—we did not define the semantics so far—

1: $x_1 := 1$
2: while $x_1 \neq 0$ do
3: $x_1 := 1$
4: od

is such a program.\(^1\) The while loop never terminates and there is no state that can be reached since there is no “after the execution of the program”.

1. If $P$ is a simple statement then

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

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 $\Phi_P^{(r)}(S)$ is either undefined, which means that $P_1$

\(^1\)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.
Lemma 2.3  
For every program \( P \) (since undefined means that \( W \) then simpler will mean that the programs are in \( W \)), let \( n \) on \( \mathbb{N} \). Induction becomes ordinary induction, since we now can just do induction on \( n \).

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 \( P \in W_n \), then simpler will mean that the programs are in \( W_{n-1} \). In this way, structural induction becomes ordinary induction, since we now can just do induction on \( n \).

Proof. Induction base: If \( P \in W_0 \), 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 \( P \in W_n \ \backslash \ W_{n-1} \) for \( n > 0 \). We assume that \( \Phi_Q \) is well-defined for all programs \( Q \in W_{n-1} \). Since \( n > 0 \), \( P \) is either \( \text{while} \ x_1 \neq 0 \ {\text{do}} \ P_1 \ {\text{od}} \) or \( P_1;P_2 \) for \( P_1,P_2 \in W_{n-1} \). By the induction hypothesis, \( \Phi_{P_1} \) and \( \Phi_{P_2} \) are well-defined. If \( P \) is of the former form, then \( \Phi_P(S) \) is also well-defined for every \( S \) by the rules in 2.(a). In the case that \( P = P_1;P_2 \) there is a caveat: There could be another decomposition of \( P = P'_1;P'_2 \) with \( P_1 \neq P'_1 \). We have to show that in both cases, \( \Phi_P \) will be the same function, i.e., that for all \( S \), \( \Phi_{P_2}(\Phi_{P_1}(S)) = \Phi_{P'_2}(\Phi_{P'_1}(S)) \) whenever both sides are defined or both sides are undefined.

By Exercise 2.2, \( P \) can be uniquely written as \( Q_1;Q_2;\ldots;Q_k \) where each \( Q_k \) is either a simple statement or a while loop. This means that there are indices \( i \) and \( j \) such that \( P = Q_1;\ldots;Q_j \) and \( P'_1 = Q_1;\ldots;Q_j \). W.l.o.g. assume that \( i > j \). This means that

\[
\Phi_{P_2}(\Phi_{P_1}(S)) = \Phi_{Q_{i+1};\ldots;Q_k}(\Phi_{Q_1;\ldots;Q_i}(S)) \\
= \Phi_{Q_{i+1};\ldots;Q_k}(\Phi_{Q_{i+1};\ldots;Q_j}((\Phi_{Q_1;\ldots;Q_j}(S)))) \\
= \Phi_{Q_{j+1};\ldots;Q_k}(\Phi_{Q_1;\ldots;Q_j}(S)) \\
= \Phi_{P'_2}(\Phi_{P'_1}(S))
\]

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

3What 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.
whenever all arising terms are defined. If one of them is not defined, then both $\Phi_{P_2}(\Phi_{P_1}(S))$ and $\Phi_{P_2'}(\Phi_{P_1'}(S))$ are not defined, which is exactly what we want. In the equation above, we used the induction hypothesis two times.

Exercise 2.2 Every WHILE program $P$ can be written as $P = P_1; P_2; \ldots; P_k$ such that each $P_\kappa$, $1 \leq \kappa \leq k$, is either a simple statement or a while loop. This decomposition of $P$ is unique.

Remark 2.4 We could have circumvented the problem above by requiring in rule 2.(b) that if $P = P_1; P_2$, then $P_1$ is either a simple statement or a while loop. This avoids the ambiguity above. Or we could have used brackets, i.e., rule 2.(b) would have been replaced by $P = [P_1; P_2]$.

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.

The semantics of FOR programs is defined in the same manner. 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 $P_1$ $\sigma_i$ times, where $\sigma_i$ 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. 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 $\Phi$.

\footnote{We call a program a while loop if it is of the form while $x_i \neq 0$ do $Q$ od for some WHILE program $Q$.}

\footnote{For loops in C++, for instance, work differently; they are merely while loops.}
Exercise 2.3 Show that every FOR loop can be simulated by a WHILE loop. (Simulation here means that for every FOR program \( P \) of the form \textbf{for} \( x_i \) \textbf{do} \( P_1 \) \textbf{od} we can find a WHILE program \( Q \) such that \( \Phi_P = \Phi_Q \), i.e., both programs compute the same function.)

2.3 Computable functions and sets

Definition 2.5 Let \( P \) be a WHILE program. The function \( \varphi_P : \mathbb{N}^s \to \mathbb{N} \) computed by \( P \) is defined by

\[
\varphi_P(\alpha_1, \ldots, \alpha_s) = \begin{cases} 
\text{first entry of } \Phi_P((\alpha_1, \ldots, \alpha_s, 0, \ldots, 0)) & \text{if } \Phi_P((\alpha_1, \ldots, \alpha_s, 0, \ldots, 0)) \text{ is defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

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

Definition 2.6
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 \( \mathcal{R} \).
4. The class of all FOR computable functions is denoted by \( \mathcal{PR} \).

The acronyms \( \mathcal{R} \) and \( \mathcal{PR} \) 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 this lecture.

By Exercise 2.3, \( \mathcal{PR} \subseteq \mathcal{R} \). Since for loops always terminate, all functions in \( \mathcal{PR} \) 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 \( \mathcal{R} \setminus \mathcal{PR} \). 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. 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 \( \chi_L \) as a function \( \mathbb{N} \to \mathbb{N} \).

Definition 2.7
1. A language \( L \subseteq \mathbb{N} \) is called recursive or decidable if \( \chi_L \in \mathcal{R} \).
2. The set of all recursive languages is denoted by \( \mathcal{REC} \).

© Markus Bläser 2007–2015
3 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 $\mathbb{N}^s \rightarrow \mathbb{N}$ that JAVA or C++ could compute. (And no, we are not able to display fancy graphics.)

3.1 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.

3.2 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.

3.3 Procedures and functions

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

Above, $\ell$ 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 occurrence 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.

Exercise 3.1 The evaluation strategy above is call by value. How do you implement call by reference?
### 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. \( x_i = x_{\ell+1} \)

### 3.4 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. \textbf{for} \( x_j \) \textbf{do}
3. \( x_i := x_i + x_k \)
4. \textbf{od}

It is easy to see that the program above is correct. Since a FOR loop can be simulated by a WHILE loop, we can perform multiplications in WHILE, too.

**Exercise 3.2** 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 \( x_i := x_j \text{ rem } x_k \) in ordinary FOR.

**Exercise 3.3** Show that we even could replace the simple statements \( x_i := x_j + x_k \), \( x_i := x_j - x_k \), and \( x_i := c \) by the three statements \( x_i++ \), \( x_i-- \), and \( x_i := 0 \). \( x_i++ \) increases the content of \( x_i \) by 1. \( x_i-- \) decreases it by 1 except when the content of \( x_i \) equals 0, then it is left unchanged.

### 3.5 Arrays

Built-in types like \texttt{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 \texttt{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.

**Lemma 3.1** There are FOR computable functions \( \langle \cdot, \cdot \rangle : \mathbb{N}^2 \to \mathbb{N} \) and \( \pi_i : \mathbb{N} \to \mathbb{N}, i = 1, 2 \), such that

\[ \pi_i(\langle x_1, x_2 \rangle) = x_i \]

© Markus Bläser 2007–2015
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. \(\langle \cdot, \cdot \rangle\) is obviously FOR computable by the results in Section 3.4 and Exercise 3.3.

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 3.4, we construct FOR programs for \(\pi_1\) and \(\pi_2\).

**Exercise 3.4** Construct FOR programs for the projections \(\pi_i, i = 1, 2\).

**Remark 3.2** \(\langle \cdot, \cdot \rangle\) is also surjective.

While you can forget about the rest of the construction of 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**

Karl 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: 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.

Now if we want to store \(a_0, \ldots, a_{n-1} \in \mathbb{N}\) in one number, we store them as \(\langle a_{n-1}, \langle a_{n-2}, \ldots, \langle a_2, \langle a_1, a_0 \rangle \ldots \rangle \rangle\). Assume that \(A\) is the variable where we want to store our array and \(n\) contains the number of elements. (We are already using the convention from Section 3.1.) Initially, all entries are zero. It is easy to check that \(\langle 0, \ldots, \langle 0, (0,0) \ldots \rangle \rangle = 0\). Thus initially, we set \(A := 0\). We want to extract the \(i\)th element from \(A\) and can do this as shown in Program 3. This program is not a WHILE/FOR program not even if you apply all the syntactic sugar mentioned so far. But I am sure that you

---

\(^{1}\)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.
Program 3 Extracting the ith element from $A$

Output: $A[i]$ is returned in $x_0$

1: $x_0 := A$;
2: for $n - i - 1$ do
3:   $x_0 := \pi_2(x_0)$
4: od;
5: if $i \neq 0$ then
6:   $x_0 := \pi_1(x_0)$
7: fi

can convert it into a true WHILE/FOR program. The idea of the algorithm above is simple. If we want to extract $a_i$ for $i > 0$, then we will get it via

$$
\pi_1 \circ \pi_2 \circ \cdots \circ \pi_2(A).
$$

If $i = 0$, then we will get $a_i$ via

$$
\pi_2 \circ \cdots \circ \pi_2(A).
$$

Exactly this is implemented in the program above.

Now assume that we want to store the value of $b$ in the $i$th place of $A$. The idea is basically the same as before, but now we also have to restore the array. Program 4 shows how to achieve this.

Program 4 Simulating $A[i] := b$

1: $x_0 := A$;
2: $x_1 := 0$;
3: for $n - i - 1$ do
4:   $x_1 := \langle \pi_1(x_0), x_1 \rangle$;
5:   $x_0 := \pi_2(x_0)$
6: od;
7: if $i \neq 0$ then
8:   $x_0 := \langle b, \pi_2(x_0) \rangle$
9: else
10:   $x_0 := b$
11: fi
12: for $n - i - 1$ do
13:   $x_0 := \langle \pi_1(x_1), x_0 \rangle$;
14:   $x_1 := \pi_2(x_1)$
15: od
16: $A := x_0$;
In the first for loop, we find the position of the element that we want to change as we did before. In addition, we also store the elements that we remove from \( x_0 \) in \( x_1 \). Assume that, for instance, \( x_0 \) equals \( \langle a_i, \langle a_{i-1}, \ldots, \langle a_2, \langle a_1, a_0 \rangle \ldots \rangle \rangle \ldots \rangle \). We replace \( a_i \) by \( b \). If \( i = 0 \), then \( a_i \) is the remaining element in \( x_0 \) and we can just overwrite it with \( b \). If \( i > 0 \), then we throw \( a_i \) away by computing \( \pi_2(x_0) \) and replace it by \( b \) via the pairing function. Note the 0 appended to the end of \( x_1 \). This makes a case distinction redundant when reconstructing \( A \) in the second for loop; we can always extract the next element from \( x_1 \) by applying \( \pi_1 \). (One could also make this convention for \( A \).) In this second loop, we insert the elements that were removed from \( x_0 \) back into \( x_0 \). We extract them one by one from \( x_1 \) and add them to \( x_0 \) by using the pairing function.

**Exercise 3.5** 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, this will result in an error. So it works like a stack of plates where you can only either remove the top plate or put another plate on the top.\(^2\) There is usually also a function isempty that allows you to check whether a stack is empty or not.

1. How do you store a stack of natural numbers in one natural number?
2. Use the pairing function to implement the push operation in WHILE.
3. Use the projections to implement the pop operation in WHILE.
4. Implement the isempty function.

### 3.6 Further exercises

**Exercise 3.6** Show how to simulate the if-then-else statement in FOR.

**Exercise 3.7** The following two statements are also useful. Explain how to simulate them in simple WHILE.

1. **Input:** \( v_1, \ldots, v_s \) declares \( v_1, \ldots, v_s \) as the input variables.
2. **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 functions. It is injective but not surjective.

\(^2\)Dexterous people might do different things but computer scientist usually do not.
Exercise 3.8  Let $k$ be some constant. Let $p_1, \ldots, p_k$ be different prime numbers.

1. Show that the mapping $g$ given by

$$\mathbb{N}^k \to \mathbb{N}$$

$$(x_1, \ldots, x_k) \mapsto p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k}$$

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 $\text{im} \, g$.

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

Chapters that are numbered with latin characters instead of numbers 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.

A.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 the simplest example of a well defined 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

\[
\begin{align*}
 a(0, y) &= y + 1 \\
 a(x, 0) &= a(x - 1, 1) \\
 a(x, y) &= a(x - 1, a(x, y - 1)) 
\end{align*}
\]

for $x > 0$, $y > 0$.

Lemma A.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 $x, y \in \mathbb{N}$. Starting with the induction base $x = 0$ gives us $a(0, y) = y + 1$ for all $y$. The induction step $x - 1 \rightarrow x$ 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 A.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 5 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.

A.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

\[
\begin{align*}
a(1, y) &= a(0, a(1, y - 1)) \\
&= a(2, y - 1) + 1 \\
&= a(0, a(1, y - 2)) + 1 \\
&= a(1, y - 2) + 2 \\
&\vdots \\
&= a(1, 0) + y \\
&= a(0, 1) + y \\
&= y + 2.
\end{align*}
\]

For \( x = 2 \), we have

\[
\begin{align*}
a(2, y) &= a(1, a(2, y - 1)) \\
&= a(2, y - 1) + 2 \\
&\vdots \\
&= a(2, 0) + 2y \\
&= a(1, 1) + 2y \\
&= 2y + 3.
\end{align*}
\]
Finally, for $x = 3$, we obtain

\begin{align*}
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) + 32 \\
&= 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^k \\
&= 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 2^4 - 3 \\
&= 2^{y+3} - 3.
\end{align*}

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

### A.3 Some useful facts

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

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

\[ y < a(x, y). \]

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

\[ a(0, y) = y + 1 > y. \]

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

\[ a(x, 0) = a(x - 1, 1) > 1 > 0, \]

© Markus Bläser 2007–2015
where the first inequality follows from the induction hypothesis for $x$. The
induction step for the inner induction is shown as follows:

\begin{equation}
  a(x, y) = a(x-1, a(x, y-1)) > a(x, y-1) > y-1. \quad (A.1)
\end{equation}

$a(x, y) = a(x-1, a(x, y-1))$ 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 (A.1), there stands a “$>$”
twice. Thus (A.1) even implies $a(x, y) > y$. ■

**Lemma A.4** The Ackermann function is strictly monotonic 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 A.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 A.5** For all $x, y \in \mathbb{N}$, $a(x, y + 1) \leq a(x + 1, y)$.

*Proof.* By induction on $y$. For $y = 0$ the equation $a(x, 1) = a(x + 1, 0)$
follows from the definition of $a$.

Consider the induction step $y - 1 \to y$. By Lemma A.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 A.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). \quad \text{\(\blacksquare\)}
\]

**Lemma A.6** The Ackermann function is strictly monotonic 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 A.4 first and then Lemma A.5, we obtain

\[
a(x, y) < a(x, y + 1) \leq a(x + 1, y), \quad \text{\(\blacksquare\)}
\]
A.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

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

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 A.7 For every FOR program $P$ there is a $k$ 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 $x_i := 0$. (This was an exercise.)

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 a 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 $f_{\hat{P}}(n)$.

Let $P = x_i++$ or $P = x_i--$ 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 $P = x_i := 0;$, then the size of the output cannot be larger than the size of the output. Hence

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

too. This finishes the induction base.

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 in $P_2$. Thus

$$f_P(n) \leq a(k_1,n) + a(k_2,n) = a(k,n),$$

for some $k$. This finishes the induction step.

© Markus Bläser 2007–2015
variables before the execution of $P_2$. Altogether,

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

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 $\nu_0, \ldots, \nu_t$ for which the maximum is attained in the definition of $x_i$. Let $m := \nu_i$ be the value of $x_i$ in this tuple. We distinguish three cases:

$m = 0$: In this case, $f_P(n) = n < a(0, n)$.

$m = 1$: Here, $f_P(n) \leq f_{P_1}(n) < a(k_1, n)$.

$m > 1$: Here we have,

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

**Theorem A.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 A.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. ■
A.4. The Ackermann function is not FOR computable

Program 5 Ackermann function

Input: $x, y$

1: push($S, x$);
2: push($S, y$);
3: while size($S$) > 1 do
4:   $y := \text{pop}(S)$;
5:   $x := \text{pop}(S)$;
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: $x_0 := \text{pop}(S)$;
4 Gödel numberings

We will deal with two fundamental questions that we will use in the following chapters. 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 4.1** A set \( S \) is countable if there exists an injective\(^1\) function \( S \rightarrow \mathbb{N} \). If there is a bijection \( S \rightarrow \mathbb{N} \), then \( S \) is called countably infinite.

**Exercise 4.1** If \( S \) is countable, then \( S \) is finite or countably infinite.

(Remark: Let \( f : S \rightarrow \mathbb{N} \) an injective function. If \( S \) is not finite, then its image under \( f \) is infinite. Let \( I = f(S) \). We can now construct a bijection \( g \) by letting \( g(0) = \min\{n \mid n \in I\} \), \( g(1) = \min\{n \mid n \in I \setminus \{g(0)\}\} \), and so on. Since \( I \) is infinite, \( g(i) \) is defined for every \( i \in \mathbb{N} \). This construction bases on the fact that the natural numbers are well-ordered: every nonempty subset has a minimal element. For larger sets, this is not so clear any more. But also beyond the scope of this lecture ...)

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

**Exercise 4.2** 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 injective function \( \text{göd} : \mathcal{W} \rightarrow \mathbb{N} \), that is, different WHILE programs get different numbers. But this is not enough, we also need the following two properties:

\(^1\)Recall that injective includes that the function is total
1. There is a WHILE program $C$ that computes the characteristic function of $\text{im g"{o}d}$, and

2. There is a WHILE program $U$ that given $i \in \text{im g"{o}d}$ and $x \in \mathbb{N}$ computes $\varphi_P(x)$ where $P = \text{g"{o}d}^{-1}(i)$.

Furthermore, given $i \in \text{im g"{o}d}$ it is “easy” to find the WHILE program $P$ with $\text{g"{o}d}(P) = i$. “Easy” is not formally specified here. We could extend the WHILE language and add a print command that can print fixed string and the content of variables. Then it is quite easy (at least if you implemented a stack in WHILE) to print $P$ if $i$ is given.

Such a mapping g"{o}d is usually called Gödel numbering, named after the mathematician Kurt Gödel. The term “Gödel numbering” is used for many different things, so be careful. (Often, the existence of $U$ is not included in the definition.)

**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 arithmetic of $\mathbb{N}$, there are theorems that are true but cannot be deduced from the axioms. (More on 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 g"{o}d for WHILE programs that we will use throughout this lecture. g"{o}d will be defined inductively. We start with the simple statements:

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

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

1. If $P = \text{while } x_i \neq 0 \text{ do } P_1 \text{ od}$, then $\text{g"{o}d}(P) = \langle 3, (i, \text{g"{o}d}(P_1)) \rangle$.
2. If $P = P_1; P_2$ where $P_1$ is a while loop or a simple statement, then $\text{g"{o}d}(P) = \langle 4, (\text{g"{o}d}(P_1), \text{g"{o}d}(P_2)) \rangle$.

**Lemma 4.2** g"{o}d is well-defined.

© Markus Bläser 2007–2015
Proof. The proof is again by structural induction.

Induction base: Obviously, \( \text{göd}(P) \) is well-defined for all \( P \in W_0 \).

Induction step: Now assume that \( P \in W_n \) for some \( n \). Then either \( P = \textbf{while } x_i \neq 0 \textbf{ do } P_1 \textbf{ od } \) or \( P = P_1; P_2 \) for some \( P_1, P_2 \in W_{n-1} \). In the latter case, \( P_1 \) is a while loop or a simple statement. By the induction hypothesis, \( \text{göd}(P_1) \) and \( \text{göd}(P_2) \) are both well-defined. Thus \( \text{göd}(P) = \langle 3, \langle i, \text{göd}(P_1) \rangle \rangle \) or \( \text{göd}(P) = \langle 4, \langle \text{göd}(P_1), \text{göd}(P_2) \rangle \rangle \), respectively, are both well-defined, too.

For the latter case, note that the decomposition into \( P_1; P_2 \) is unique by Exercise 2.2. ■

Lemma 4.3 \( \text{göd} \) is injective.

Proof. We show the statement that for all \( n \), \( \text{göd}(P) = \text{göd}(Q) \) implies \( P = Q \) for all \( P, Q \in W_n \). From this, the assertion of the lemma follows.

Induction base: The statement is clear for all programs in \( W_0 \). This shows the induction base.

Induction step: Now assume that \( \text{göd}(P) = \text{göd}(Q) \) and assume that \( P \in W_n \setminus W_{n-1} \) for some \( n > 0 \) and \( Q \in W_n \). Since \( n > 0 \), \( \text{göd}(P) \) is either \( \langle 3, \langle i, \text{göd}(P_1) \rangle \rangle \) or \( \langle 4, \langle \text{göd}(P_1), \text{göd}(P_2) \rangle \rangle \) for some programs \( P_1, P_2 \in W_{n-1} \). We only treat the case \( \text{göd}(P) = \langle 3, \langle i, \text{göd}(P_1) \rangle \rangle \), the other case is an exercise. \( \text{göd}(P) = \text{göd}(Q) \) in particular implies that \( \pi_1(\text{göd}(P)) = \pi_1(\text{göd}(Q)) \).

This shows that \( \text{göd}(Q) = \langle 3, \langle j, \text{göd}(Q_1) \rangle \rangle \). 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 \( \text{göd}(P_1) = \text{göd}(Q_1) \). By the induction hypothesis, \( P_1 = Q_1 \). Thus \( P = Q \). ■

Corollary 4.4 The set of all WHILE programs \( W \) is countable.

For the rest of this lecture, we denote the image of \( \text{göd} \) by \( \mathcal{G} \).

Exercise 4.3 Fill in the missing part in the proof of Lemma 4.3.

Excursus: Programming systems

The mapping \( \text{göd} \) associates with every \( i \in \mathcal{G} \) a function \( \varphi_{\text{göd}^{-1}(i)} \). Instead of \( \varphi_{\text{göd}^{-1}(i)} \) we write \( \varphi_i \). If we associate with every \( i \notin \mathcal{G} \) a fixed function \( \varphi_i \), say the function that is undefined everywhere, we get an infinite sequence of functions \( \langle \varphi_i \rangle_{i \in \mathbb{N}} \).

Definition 4.5 1. A sequence \( \langle \psi_i \rangle_{i \in \mathbb{N}} \) is called a programming system if the set of all \( \psi_i \) is precisely \( \mathbb{R} \), 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 \( \psi_u((j, x)) = \psi_j(x) \) for all \( j, x \in \mathbb{N} \).

© Markus Bläser 2007–2015
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 \( j,k \in G \). If \( j \) or \( k \) is not in \( G \), then \( \varphi_c \) just outputs the index of the function that is undefined everywhere. We can check whether \( j,k \in G \) once we constructed the program \( C \).

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.)
In this chapter we will answer the question whether there is a function $\mathbb{N} \rightarrow \mathbb{N}$ that is not computable by a WHILE program.

5.1 Proof by “counting”

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

Theorem 5.1 The set of all total functions $\mathbb{N} \rightarrow \{0, 1\}$ 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 $\mathbb{N} \rightarrow \{0, 1\}$, 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 $c : \mathbb{N} \rightarrow \{0, 1\}$ 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 $n : F \rightarrow \mathbb{N}$ be a bijection. 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 $i$th row contains the values of $f_i$ and the $j$th 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 5.1).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$f_0(0)$</td>
<td>$f_0(1)$</td>
<td>$f_0(2)$</td>
<td>$f_0(3)$</td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>$f_1(0)$</td>
<td>$f_1(1)$</td>
<td>$f_1(2)$</td>
<td>$f_1(3)$</td>
<td>…</td>
</tr>
<tr>
<td>2</td>
<td>$f_2(0)$</td>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>…</td>
</tr>
<tr>
<td>3</td>
<td>$f_3(0)$</td>
<td>$f_3(1)$</td>
<td>$f_3(2)$</td>
<td>$f_3(3)$</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Figure 5.1: The diagonalization scheme
5.1. Proof by “counting”

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 5.1. Clearly $c \in F$. But this means that there is an index $i_0$ such 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 5.2** 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 $W$. Since the set $W$ of all WHILE programs is countable, there is an injective mapping $i_2$ from $W \to \mathbb{N}$. 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 5.3** 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 creator 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:


Cantor should be right.
5.2 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 $i \in \mathcal{G}$, $c(i)$ 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 5.2: We will use the same diagonalization scheme as in Figure 5.1. 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 5.2. We constructed an injective mapping $\text{göd}$ from the set of all WHILE programs to $\mathbb{N}$ in Chapter 4. We now define a sequence of functions $f_0, f_1, f_2, \ldots$ by

$$f_i(j) = \begin{cases} \varphi_P(j) & \text{if } i \text{ encodes the program } P, \text{ i.e., } \text{göd}(P) = i \\ 0 & \text{if } i \text{ does not encode any WHILE program} \end{cases}$$

for all $i$ and $j$. That means, if $i$ encodes a WHILE program, i.e., there is a program $P$ such that $\text{göd}(P) = i$, then $f_i$ will be the function $\varphi$ computed by $P$. If $i$ does not encode a WHILE program, then $f_i$ will be a dummy function; here we choose the function that is zero everywhere.

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 $\varphi_P$ at the input $i$.

Remark 5.4 $c$ 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 5.2 holds for all programming systems. All that we used is that there is a mapping $i \rightarrow \psi_i$ such that the image of this mapping is $\mathbb{R}$. 

© Markus Bläser 2007–2015
5.3 Further exercises

Exercise 5.1 Show that for any nonempty set $S$, there is no bijection between $S$ and its power set $\mathcal{P}(S)$. 
6 A universal WHILE program

In this chapter, we will construct the WHILE programs $C$ and $U$ for our function $\text{göd}$. Assume that we are given an index $g \in \text{imgöd}$, i.e., a valid 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 $\ell$ 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 $\ell$ given $g$. But since any upper bound on $\ell$ is fine too, we just use an array of length $g$ in $U$. $g$ is an upper bound on $\ell$ because of Exercise 6.1 (and the way we constructed $\text{göd}$).

Exercise 6.1 Show that for all $j,k \in \mathbb{N}$, $\langle j,k \rangle \geq \max\{j,k\}$.

A simple statement is encoded as $\langle 0, \langle i, \langle j,k \rangle \rangle \rangle$ (addition), $\langle 1, \langle i, \langle j,k \rangle \rangle \rangle$ (subtraction), or $\langle 2, \langle i, c \rangle \rangle$ (initialization with constant). Using $\pi_1$, 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 $\pi_2$ then gives us the information about the variables and/or constants involved. Program 6 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 6 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 6.2 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 from Exercise 3.5 how to implement a stack in WHILE (and even FOR).
Program 7 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 \( P := \text{gcd}^{-1}(g) \).

\( S \): the stack that stores (encodings of) pieces of \( P \) to be executed later

\( \text{cur} \): a variable that stores the piece of \( P \) that we are right now simulating

\( \text{term} \): is 0 if our simulation terminated and 1 otherwise.

\( \text{type} \): stores the type (0 to 4) of the current statement that we are simulating

In lines 1 to 5, we initialize \( X \) 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 \( P \). \( \text{cur} \) will always be of the form

- \( \langle 0, \langle i, \langle j, k \rangle \rangle \rangle \) (addition)
- \( \langle 1, \langle i, \langle j, k \rangle \rangle \rangle \) (subtraction)
- \( \langle 2, \langle i, c \rangle \rangle \) (initialization)
- \( \langle 3, \langle i, \text{gcd}(P_1) \rangle \rangle \) (while loop)
- \( \langle 4, \langle \text{gcd}(P_1), \text{gcd}(P_2) \rangle \rangle \) (concatenation)

In line 6, we set \( \text{type} := \pi_1(\text{cur}) \). This value is between 0 to 4. If \( \text{type} \in \{0, 1, 2\} \), then we just have to simulate an addition, subtraction, or initialization. This is easy. The next two cases are far more interesting.

If \( \text{type} = 3 \), then \( \text{cur} = \langle 3, \langle i, \text{gcd}(P_1) \rangle \rangle \). So we have to simulate a while loop \( \text{while } x_i \neq 0 \text{ do } P_1 \text{ od} \). 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 \( \text{cur} \) (which equals \( \langle 3, \langle i, \text{gcd}(P_1) \rangle \rangle \)) and \( \pi_2(\pi_2(\text{cur})) \) (which equals \( \text{gcd}(P_1) \)) on the stack.

If \( \text{type} = 4 \), then \( \text{cur} = \langle 4, \langle \text{gcd}(P_1), \text{gcd}(P_2) \rangle \rangle \). We push \( \pi_2(\pi_2(\text{cur})) \) (which is \( \text{gcd}(P_2) \)) on the stack and then \( \pi_1(\pi_2(\text{cur})) \) (which is \( \text{gcd}(P_1) \)). 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 \( \text{cur} \), and go on with the simulation.

\footnote{That is not 100% correct; what we mean is that the value of \( \text{type} \) is in \( \{0, 1, 2\} \). We will use this sloppy notation often in the remainder of this chapter.}
**Program 7** 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: $S := 0$;
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 6).
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($cur, S$); {push the loop once again}
21:         push($\pi_2(\pi_2(cur)), S$) {push the body of the loop}
22:    fi
23: fi
24: if $type = 4$ then
25:     push($\pi_2(\pi_2(cur)), S$); {push two parts onto stack in reverse order}
26:     push($\pi_1(\pi_2(cur)), S$)
27: fi
28: if isempty($S$) = 0 then
29:     $cur := \text{pop}(S)$
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 6.2 is a special case of it.

**Lemma 6.1** Let $T$ be the state that corresponds to the content of $X$, let $\sigma$ be the content of stack $S$, and let $P = \text{göd}^{-1}(\text{cur})$ 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 26 at the end of the while loop is again $\sigma$ for the first time. Then $T' = \Phi_P(T)$, if this event occurs. If $\sigma$ 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 6.3** Prove Lemma 6.1. You can use—ta dah!—structural induction.

**Theorem 6.2** There is a WHILE program $U$ that given $g \in \text{im göd}$ 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 6.3 (Kleene normal form)** Let $f$ be a WHILE computable function. Then there are FOR program $P_1$, $P_2$, and $P_3$ such that the program

\[
P_1; \text{while } x_1 \neq 0 \text{ do } P_2 \text{ od; } P_3
\]

computes $f$

**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. □

**Exercise 6.4** Modify the program $U$ in such a way that we get the program $C$ that checks whether a given $g$ is in $\text{im göd}$. Can you achieve that $C$ is a FOR program?

### 6.1 Further exercises

**Exercise 6.5** 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 $\varphi_P(x)$?

**Exercise 6.6** Show that the following function is FOR computable:

\[
(i, (x, t)) \mapsto \begin{cases} 
1 & \text{if program } \text{göd}^{-1}(i) \text{ halts on } x \text{ after } \leq t \text{ steps} \\
0 & \text{otherwise}
\end{cases}
\]

“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.

© Markus Bläser 2007–2015
7 The halting problem

The halting problem \( H \) is the following problem:

\[
H = \{ (g, m) \mid g \in \text{im } \text{göd} \text{ and } \text{göd}^{-1}(g) \text{ halts on } m \}.
\]

This is a natural problem. The special halting problem \( H_0 \) is the following special case:

\[
H_0 = \{ g \mid g \in \text{im } \text{göd} \text{ and } \text{göd}^{-1}(g) \text{ halts on } g \}.
\]

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.

7.1 The halting problem is not decidable

**Theorem 7.1** \( H_0 \notin \text{REC} \), 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 \( x_1 \neq 0 \) 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 \( \text{göd}(P) \). In this case, \( P_0 \) returns 0 on \( \text{göd}(P) \). But this means, that \( P \) does not terminate on \( \text{göd}(P) \), a contradiction.

If \( P \) does not terminates on \( \text{göd}(P) \), then \( P_0 \) returns 1 on \( \text{göd}(P) \). But this means that \( P \) terminates on \( \text{göd}(P) \), again a contradiction.
Since $P$ either terminates on $\text{göd}(P)$ 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 independently 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](http://www.turingarchive.org)

---

### 7.2 Recursively enumerable languages

**Definition 7.2**

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 $\text{RE}$.

**Remark 7.3** 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.

**Theorem 7.4** The halting problem and the special halting problem are recursively enumerable.

**Proof.** Let $(g, m)$ be the given input. First we use the WHILE program $C$ to check whether $g \in \text{im göd}$. If not, then we enter an infinite loop. If yes, then we simulate $g$ on $m$ using the universal WHILE program $U$. It is easy to see that this program terminates if and only if $g$ encodes a WHILE program and $\text{göd}^{-1}(g)$ halts on $m$. If it terminates, then we return 1.
Remark 7.5 The set that corresponds to the characteristic function $c$ constructed in the alternative proof of Corollary 5.2 is not recursively enumerable, since we diagonalized against all WHILE programs not only those that compute total functions.

Theorem 7.6 The following two statements are equivalent:

1. $L \in \text{REC}$.
2. $L, \overline{L} \in \text{RE}$.

Proof. For the “$\Rightarrow$” direction, note that $L \in \text{REC}$ implies $\overline{L} \in \text{REC}$ and that $\text{REC} \subseteq \text{RE}$.

For the other direction, note that there are WHILE programs $P$ and $\overline{P}$ that halt on all $m \in L$ and $m \notin L$. So either $P$ or $\overline{P}$ 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 $m \in \overline{L}$ and we never have a chance to run $\overline{P}$ on $m$.

The trick is to run $P$ and $\overline{P}$ 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 $\overline{P}$. (We need two stacks $S_1, S_2$ to do this, two instances $\text{cur}_1, \text{cur}_2$ of the variable $\text{cur}$, etc.) Eventually, one of the programs $P$ or $\overline{P}$ will halt. Then we know whether $m \in L$ or not. ■

Corollary 7.7 $\overline{H_0} \notin \text{RE}$.

Proof. We know that $H_0$ is in $\text{RE}$ but not in $\text{REC}$. By Theorem 7.6, $\overline{H_0}$ is not in $\text{RE}$. ■

Exercise 7.1 Show that the following three statements are equivalent:\footnote{This explains the name recursively enumerable: There is a WHILE computable function, here $\varphi_P$, that enumerates $L$, that means, if we compute $\varphi_P(0), \varphi_P(1), \varphi_P(2), \ldots$, we eventually enumerate all elements of $L$.}

1. $L \in \text{RE}$.
2. There is a WHILE program $P$ such that $L = \text{im } \varphi_P$.
3. $L = \emptyset$ or there is a FOR program $P$ such that $L = \text{im } \varphi_P$. 
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 \( i, j \in \text{im} \, \text{göd} \), 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 artificial 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 a 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, \( V, V_0, T \notin \text{REC} \). 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 \text{REC}, too.

### 8.1 Many-one reductions

**Definition 8.1** 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

\[
\text{for all } x \in \mathbb{N}: \ x \in L \iff f(x) \in L'.
\]

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 8.2** If \( P \) were a program for \( H \) (we already know that it does not exist), then the following \( P_0 \) would be a program for \( H_0 \):

\[
\begin{align*}
1: \ x_0 & := \langle x_0, x_0 \rangle; \\
2: \ P
\end{align*}
\]

The first line prepares the input for \( P \). \( H_0 \) expects a Gödel number, say \( g \), and the first line forms the pair \( \langle g, g \rangle \), the input for \( H \). We have \( g \in H_0 \) iff \( \langle g, g \rangle \in H \). Questions about membership in \( H_0 \) reduce to questions about membership in \( H \).

The second important property of reductions is that they should be “easy”: If \( g \) is the Gödel number of \( P \), then \( \langle 4, \langle \text{göd}(x_0 := \langle x_0, x_0 \rangle), g \rangle \rangle \) is the Gödel number of \( P_0 \). Note that \( \text{göd}(x_0 := \langle x_0, x_0 \rangle) \) is a constant that can be hard-wired into the reduction.

The arguments above show \( H_0 \leq H \).

The next lemma shows the importance of reductions. If \( L \leq L' \), then the fact that \( L' \) is easy (that means, is in \( \text{RE} \) or \( \text{REC} \)) implies that \( L \) is easy, too. The contraposition of this is: If \( L \) is hard (that is, is not in \( \text{RE} \) or \( \text{REC} \)) then \( L' \) is also hard.

**Lemma 8.3** Let \( L, L' \subseteq \mathbb{N} \) and \( L \leq L' \). Then the following statements hold:

1. If \( L' \in \text{RE} \), then \( L \in \text{RE} \).
2. If \( L' \in \text{REC} \), then \( L \in \text{REC} \).

**Proof.** Let \( f \) be a many one reduction from \( L \) to \( L' \). We prove the first statement. Since \( L' \in \text{RE} \), 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 \( P \):

\[
\begin{align*}
1: \ x_0 & := f(x_0) \\
2: \ P'
\end{align*}
\]

It computes \( \varphi_{P} = \varphi_{P'} \circ f \). We have

\[
x \in L \implies f(x) \in L' \implies \varphi_{P'}(f(x)) = 1 \implies \varphi_{P}(x) = 1
\]
8.1. Many-one reductions

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 \( L \in \text{RE} \) (as witnessed by \( P \)). The second statement is shown in the same fashion, we just set \( \varphi_{P'}(m) = 0 \) if \( m \notin L' \) and have to replace “is undefined” in the second series of implications by “= 0”.

**Corollary 8.4** Let \( L, L' \subseteq \mathbb{N} \) and \( L \leq L' \). Then the following statements hold:

1. If \( L \notin \text{RE} \), then \( L' \notin \text{RE} \).
2. If \( L \notin \text{REC} \), then \( L' \notin \text{REC} \).

**Proof.** These statements are the contrapositions of the statements of the previous lemma.

---

### 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 8.3 is not true for Turing reductions (which is a fancy name for arbitrary subroutine use).

---

**Lemma 8.5** \( \leq \) 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 \( g \circ f \).

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 8.1** 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 8.1.)
Figure 8.1: 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 $N$ that are not in $L'$. In short, “yes”-instances go to “yes”-instances and “no”-instances to “no”-instances.

8.2 Termination and Verification

We show that neither $V_0$ nor $V$ nor $\bar{V}_0$ nor $T$ are decidable.

Lemma 8.6 $H_0 \leq V_0$.

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, $\varphi_{Q_i}$ is the function that is nowhere defined. In other words,

$$\text{göd}(P) \in H_0 \iff \text{göd}_{TM}(Q_i) \in V_0.$$ 

Thus the mapping $f$ that maps

$$i \mapsto \begin{cases} 
\text{göd}(Q_i) & \text{if } i \in \text{im göd} \\
i & \text{otherwise}
\end{cases}$$

is the desired reduction. (The case $i \notin \text{im göd}$ is only of technical nature. If $i \notin \text{im göd}$, then $i \notin H_0$ and $i \notin V_0$.)

We only have to convince ourselves that $f$ is indeed WHILE computable. The Gödel numbers of the three parts of $Q_i$:

- $(2, (0, i))$. 

© Markus Bläser 2007–2015
Therefore, the reductions is given by

\[ i \mapsto \langle 4, \langle 2, \langle 0, i \rangle \rangle, \langle 4, \langle 2, \langle 0, 0 \rangle \rangle \rangle \rangle \] (8.1)

Note that the right hand side is not a Gödel number if \( i \) is not. (Therefore, we do not exactly compute \( f \) but something similar which does the job, too.) The above mapping is WHILE computable since the pairing function is. So the reduction \( H_0 \leq V_0 \) has an easy explicit description.

But wait, is it really this easy? Well, almost. Note that when concatenating two programs with Gödel numbers \( j \) and \( k \), then the corresponding Gödel number is \( \langle 4, \langle j, k \rangle \rangle \). But this is only defined if the program corresponding to \( j \) is a simple statement or a while loop. So if \( P \) above is not a simple statement or a while loop, this reduction is not correct.

There are two ways to solve this. Either we can parse \( P \) and restructure the whole program. This can be done in using a stack like the programs \( C \) and \( U \) do. Or we can wrap \( P \) into a while loop that is executed exactly once:

1: \( x_i := 1; \)
2: \textbf{while} \( x_i \neq 0 \) \textbf{do}
3: \( x_i := 0; \)
4: \( P \)
5: \textbf{od}

Note that the variable \( x_i \) cannot appear in \( P \) since \( i \) is the Gödel number of \( P \). This works because our paring function increases monotonically and every Gödel number contains the indices of the variables occurring in the corresponding program. In the new program, we now either concatenate programs where the first statement is a simple statement or a while loop. So the final reduction is given by an expression similar to (8.1), more precisely by

\[ i \mapsto c(\langle 2, \langle 0, i \rangle \rangle, c(\langle 2, \langle i, 1 \rangle \rangle, c(\langle 3, \langle i, c(\langle 2, \langle i, 0 \rangle \rangle, i \rangle) \rangle))) \]

where \( c(a, b) := \langle 4, \langle a, b \rangle \rangle \). Thus \( f \) is WHILE computable.\(^1\) ■

**Exercise 8.2** Show that there is a WHILE program that given two encodings \( i, j \in \text{im } \text{göd} \), constructs the Gödel number of a WHILE program that is the concatenation of \( \text{göd}^{-1}(i) \) and \( \text{göd}^{-1}(j) \). In the next chapter, we will see a more formal way to do this.

Since \( V_0 \) is a special case of \( V \), \( V_0 \) can be reduced to \( V \).

\(^1\)Often, in our reductions, we are just concatenating the given (encoding of a) WHILE program with some other encodings. This can usually be accomplished by a WHILE computable total function.
Lemma 8.7 $V_0 \leq V$.

Proof. The reduction is given by $i \mapsto (i, e)$ where $e$ is the Gödel number of a WHILE program the constant function $0$. ■

Verification is even harder than the halting problem, since $V_0$ is not recursively enumerable.

Lemma 8.8 $\bar{H}_0 \leq V_0$.

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 straightforward 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 $i \in \text{im göd}$, 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

$$i \mapsto \begin{cases} \text{göd}(K_i) & \text{if } i \in \text{im göd,} \\ y & \text{otherwise,} \end{cases}$$

where $y$ is some element in $V_0$, is a many-one reduction from $\bar{H}_0$ to $V_0$. Note that elements $i \notin \text{im göd}$ are in $\bar{H}_0$. Like before, it is easy to see that this mapping is WHILE computable. ■

Lemma 8.9 $V_0 \leq T$.

Proof. For a given $i \in \text{im göd}$, construct the following WHILE program $J_i$:

1: Simulate $P := \text{göd}^{-1}_{TM}(i)$ on $x_0$.
2: If $P$ does not output 0, then enter an infinite loop.
3: Otherwise halt.
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 either
case, $J_i$ does not terminate on this input.

Thus the mapping that maps

$$i \mapsto \begin{cases} 
gőd(J_i) & \text{if } i \in \text{im gőd} \\
i & \text{otherwise} \
\end{cases}$$

is a many-one reduction from $V_0$ to $T$. ■

Summarizing the result of this section, we get the following theorem. For
the complements of the mentioned languages, note that by Exercise 8.1, if,
say, $H_0 \leq V_0$ then $\overline{H_0} \leq \overline{V_0}$, too.

**Theorem 8.10** $V$, $V_0$, 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 reduc-
tion property.

3. "Prove" that $f$ is WHILE computable. (A formal proof of this
   might be tedious. A quick argument is usually sufficient.)
9  More on reductions

From now on, if \( i \in \text{im } \varphi \)ód, we will use \( \varphi_i \) as a synonym for \( \varphi_{\text{gód}^{-1}}(i) \); this is only done to simplify notations as a reward for working through eight chapters so far.

9.1  S-m-n Theorem

**Theorem 9.1 (S-m-n Theorem)** For every \( m, n \geq 1 \), there is a FOR computable function \( S^m_n : \mathbb{N}^{m+1} \to \mathbb{N} \) such that for all \( g \in \text{im } \text{gód}, y \in \mathbb{N}^m \), and \( z \in \mathbb{N}^n \)

\[
\varphi^{m+n}_{g}(y, z) = \varphi^{n}_{S^m_n(g, y)}(z). \quad 1
\]

Furthermore, if \( g \notin \text{im } \text{gód} \), then \( S^m_n(g, y) \notin \text{im } \text{gód} \).

**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 \( P_g = \text{gód}^{-1}(g) \) 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: \( x_m := x_0 \);

\[1\] 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 \( \varphi^{m+n}_g \) 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 \( \varphi^{m+n}_g(y, z) \) instead of formally forming a vector of length \( m + n \) out of \( y \) and \( z \) and then plugging this vector into \( \varphi^{m+n}_g \).
4: \( x_{m-1} := \eta_{m-1} \);
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.) The constructions in the last chapter were built in such a way that we automatically get that if \( g \) is not a Gödel number, then \((g, y)\) is not mapped to a Gödel number either. This mapping above is the desired mapping, since

\[
\varphi^{m+n}_g(y, z) = \varphi^{m+n}_{P_g}(y, z) = \varphi^n_{Q,g}(z) = \varphi^n_{\text{göd}(Q_{g,y})}(z) = \varphi^n_{S^n_m(g, y)}(z)
\]

for all \( y \in \mathbb{N}^m \) and \( z \in \mathbb{N}^n \).

9.2 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 \( \overline{H}_0 \leq V_0 \).

Alternative proof of Lemma 8.8. Consider the function \( f : \mathbb{N}^2 \rightarrow \mathbb{N} \) defined by

\[
f(g, m) = \begin{cases} 
0 & \text{if } g \notin \text{im \ Göd or } \text{göd}^{-1}(g) \text{ does not halt on } g \text{ after } \leq m \text{ steps} \\
1 & \text{otherwise}
\end{cases}
\]

The function \( f \) is WHILE computable, we can use \( C \) and the clocked version of the universal Turing machine \( U \). Let \( e \) be a Gödel number of \( f \). By the S-m-n Theorem,

\[
f(g, m) = \varphi^2_{g}(g, m) = \varphi_{S^1_{1}(e, g)}(m)
\]

for all \( g, m \in \mathbb{N} \). But by construction,

\[
g \in \overline{H}_0 \iff g \notin \text{im \ Göd}_{TM} \text{ or } \text{göd}^{-1}_{TM}(g) \text{ does not halt on } g \\
\iff f(g, m) = 0 \text{ for all } \mathbb{N} \\
\iff S^1_{1}(e, g) \in V_0
\]

Thus \( S^1_{1}(e, \cdot) \) is the desired many one reduction. ■
9.3 More problems

Here is another example, more of pedagogical value. Let $c \in \mathbb{N}$. Let

$$D_c = \{ g \mid g \in \text{im göd} \text{ and } |\text{dom } \varphi_{\text{göd}^{-1}(g)}| \geq c \}$$

be the set of all encodings of Turing machines that compute a function that is defined for at least $c$ different arguments. Here is potential application:

As the last assignment of the Programmierung 2 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 $D_1 \notin \text{REC}$. 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 $D_1 \in \text{RE}$, 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 9.2** For every $c \geq 1$, $H_0 \leq D_c$.

**Proof.** Consider the following function $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined by

$$f(i, x) = \begin{cases} 0 & \text{if } i \in \text{im göd} \text{ and } \varphi_i(i) \text{ is defined}, \\ \text{undef.} & \text{otherwise.} \end{cases}$$

$f$ 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 $e \in \text{im göd}$ such that $f = \varphi_e$. By the S-m-n Theorem,

$$f(i, x) = \varphi_e^2(i, x) = \varphi_{S_1^1(e, i)}(x) \quad \text{for all } i \text{ and } x.$$ 

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. \[ \Box \]

**Theorem 9.3** For every $c$, $D_c \in \text{RE}$.

**Proof.** It is sufficient to prove that $D_c = \text{dom } \varphi_P$ for some WHILE program $P$. \[ \Box \]
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:

$L \in \text{REC}$, $L$ is decidable, $L$ is recursive, the characteristic function $\chi_L$ is WHILE computable.

Also the following statements are equivalent or mean the same: $L \in \text{RE}$, $L$ is recursively enumerable, $\chi'_L$ is WHILE computable (*do not overlook the prime*!).

For a function $f : \mathbb{N} \rightarrow \mathbb{N}$, the following statements are equivalent or mean the same: $f$ is recursive (*but $f$ can be partial!*) and $f$ is WHILE computable.
10 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 $i \in \text{im g€öd}$ is in $L$ only depends on $\varphi_i$, the function computed by the machine $\text{göd}^{-1}(i)$.

10.1 Recursion Theorem

**Theorem 10.1 (Recursion Theorem)** For every WHILE computable function $f : \mathbb{N}^{n+1} \to \mathbb{N}$, there is a $g \in \text{im g€öd}$ such that

$$\varphi^{n+1}_g(z) = f(g, z) \quad \text{for all } z \in \mathbb{N}^n.$$ 

**Proof overview:** Let $f(g, z) = \varphi_e(g, z)$. Now the S-m-n Theorem states that $f(g, z) = \varphi_{S^1_n(e, g)}(z)$ for all $z$. If we now set $g = e$, then we are almost there: we have $e$ on the left-hand side and $S^1_n(e, e)$ 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 $S^1_n(y, y)$, then basically the same argument gives the desired result.

**Proof.** The function $h$ defined by

$$h(y, z) := f(S^1_n(y, y), z) \quad \text{for all } y \in \mathbb{N}, z \in \mathbb{N}^n$$

is WHILE computable. Let $e$ be a Gödel number for $h$, i.e., $h = \varphi_e$. The S-m-n Theorem implies that

$$\varphi^{n+1}_e(y, z) = \varphi^{n+1}_{S^1_n(e, y)}(z) \quad \text{for all } z \in \mathbb{N}^n.$$ 

If we now set $y = e$ and $g = S^1_n(e, e)$, we get

$$f(g, z) = f(S^1_n(e, e), z) = h(e, z) = \varphi^{n+1}_e(e, z) = \varphi^{n+1}_{S^1_n(e, e)}(z) = \varphi^{n+1}_g(z).$$

**Remark 10.2** Given the index $e$ of $h$, we can compute $g$ by a WHILE machine.

**Exercise 10.1** Show that there is a Gödel number $j$ with $\text{dom } \varphi_j = \{j\}$. 

56
10.1. Recursion Theorem

10.1.1 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

$$
\varphi_g(z) = s(g, z) = g \quad \text{for all } z \in \mathbb{N}.
$$

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.

10.1.2 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 Turing computable since we can check whether $\varphi_e(x)$ is defined by invoking $P$. 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 $\varphi_{e_0}$ were undefined, then $f(e_0, e_0) = \varphi_{e_0}(e_0) = 0$, a contradiction again. Thus $H$ cannot be decidable.

10.1.3 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 10.3** $\text{Min} \notin \text{RE}$.

**Proof.** The proof is by contradiction. Assume that $\text{Min} \in \text{RE}$. Then, since $\text{Min} \neq \emptyset$ there is a WHILE computable total function $h$ such that $\text{Min} = \text{im } h$. The function

$$
f : (g, w) \mapsto f(g, w) = \varphi_k(w) \quad \text{with } k = h(j) \text{ and } j = \min\{i \mid g < h(i)\}
$$

© Markus Bläser 2007–2015
Rice’s Theorem 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 $\text{Min}$ is infinite.

By the recursion theorem, there is a Gödel number $e$ such that

$$
\varphi_e(w) = f(e, w) = \varphi_k(w) \quad \text{for all } w \in \mathbb{N}.
$$

By the construction of $f$, $e < k$. But we also have $k \in \text{Min} = \text{im} \ h$. This is a contradiction, as $\varphi_e = \varphi_k$ and $e < k$ implies $k \notin \text{Min}$. $
$

### 10.2 Fixed Point Theorem

A **fixed point** of a function $f : \mathbb{N} \to \mathbb{N}$ is a $z_0 \in \{0, 1\}^*$ such that $f(z_0) = z_0$. Not every function has a fixed point, $z \mapsto z + 1$ is an example. But every Turing computable total function $f$ has a semantic fixed point in the sense that $z_0$ and $f(z_0)$ are the Gödel numbers of Turing machines that compute the same function.

**Theorem 10.4 (Fixed Point Theorem)** For all Turing computable total functions $f : \mathbb{N} \to \mathbb{N}$ with $\text{im} \ f \subseteq \text{im} \ g$öd and for all $n \in \mathbb{N} \setminus \{0\}$ there is an $e \in \text{im} \ g$öd 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 Turing computable since $f$ is Turing computable and total.

By the Recursion Theorem, there is an $e \in \text{im} \ g$öd$_{TM}$ such that

$$
\varphi^n_e(y) = g(e, y) = \varphi^n_{f(e)}(y) \quad \text{for all } y \in \mathbb{N}^n.
$$

### 10.3 Rice’s Theorem

**Definition 10.5 (Index set)** A language $I \subseteq \text{im} \ g$öd is an index set if

$$
\text{for all } i, j \in \text{im} \ g$öd: \quad i \in I \text{ and } \varphi_i = \varphi_j \implies j \in I.
$$

An index set $I$ is nontrivial if, in addition, $I \neq \emptyset$ and $I \neq \text{im} \ g$öd.

**Exercise 10.2** Show that $I$ is an index set if and only if there is a set $F$ of WHILE computable functions such that $I = \{ i \in \text{im} \ g$öd | $\varphi_i \in F \}$.

If an index set contains a Gödel number $i$, then it contains all Gödel numbers $j$ of Turing machines that compute the same function as $g$öd$^{-1}(i)$. In this sense, the index sets are defined by semantic properties, i.e., properties that only depend on $\varphi_i$.  

© Markus Bläser 2007–2015
10.3. Rice’s Theorem

Example 10.6 The following languages are index sets:
1. $V_0 = \{ i \in \text{im göd} \mid \varphi_i(x) = 0 \text{ for all } x \in \mathbb{N} \}$.
2. $T = \{ i \in \text{im göd} \mid \varphi_i \text{ is total} \}$
3. $D_c = \{ i \in \text{im göd} \mid |\text{dom } \varphi_i| \geq c \} \text{ for any } c \in \mathbb{N}$,
4. $\text{Mon} = \{ i \in \text{im göd} \mid \varphi_i \text{ is monotone} \}$.

All of them are nontrivial except $D_0$.

Example 10.7 The following sets are not index sets:
1. $H_0$, the special halting problem,
2. $H$, the halting problem,
3. $N_1 = \{ g \in \text{im göd} \mid g \leq 10000 \}$.

$H_0$ is not an index set, since we constructed a Gödel number $j$ with $\text{dom } \varphi_j = \{ j \}$ in Exercise 10.1. Thus $j \in H_0$ but any other Gödel number $k$ that computes the same function does not halt on its own Gödel number $k$.

$H$ is not an index set since it consists of pairs of the form $\langle g,m \rangle$, which might not be Gödel numbers. But this is of course not a formal argument, by chance all of the pairs could be Gödel numbers. We leave it as an exercise to construct a pair $\langle g,m \rangle \notin \text{im göd}$ with $g \in \text{im göd}$.

$N_1$ 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 10.8 (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 that $i \in I$ 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 Turing computable. Since $\text{im } h \subseteq \text{im göd}$, 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 Turing machines is undecidable!
10.4 Further exercises

There are nontrivial index sets that are recursively enumerable, for instance, $D_e$. Others are not, like $T$ or $V_0$. Here is a criterion that is useful to prove that an index set is not in $\text{RE}$.

**Exercise 10.3** Let $I$ be a recursively enumerable index set. Show that for all $g \in I$, there is an $e \in I$ with $\text{dom} \varphi_e \subseteq \text{dom} \varphi_g$ and $\text{dom} |\varphi_e|$ is finite.
11 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.

11.1 Arithmetic terms and formulas

Definition 11.1 Let $V = \{x_0, x_1, x_2, \ldots \}$ be a set of variables.  
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 11.2 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 $\neg F$, $(F \lor G)$, and $(F \land G)$ are arithmetic formulas.

3. If $x$ is a variable and $F$ is an arithmetic formula, then $\exists xF$ and $\forall xF$ 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 $F = (F_1 \lor F_2)$ or $F = (F_1 \land F_2)$ 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 occurrences of $x$ in $F$ are these position in $F$ that contain the symbol $x$. An occurrence of $x$ in the

\[^1\text{As usual, we will use other names for variables, too.}\]
formula $F$ is bounded if this occurrence is contained in a subformula of $F$ of the form $\exists x G$ or $\forall x G$. An occurrence that is not bounded it is called free. $F(x/n)$ denotes the formula that we get if we replace every free occurrence 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:

$$
a(n) = n \quad \text{for all } n \in \mathbb{N},
$$
$$
a(s + t) = a(s) + a(t) \quad \text{for all terms } s \text{ and } t,
$$
$$
a(s \cdot t) = a(s)a(t) \quad \text{for all terms } s \text{ and } t
$$

**Definition 11.3** 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. $F = (F_1 \lor F_2)$ is a true formula if $F_1$ or $F_2$ are true formulas.
4. $F = (F_1 \land F_2)$ is a true formulas if $F_1$ and $F_2$ are true formulas.
5. $F = \exists x F_1$ is a true formula if there is an $n \in \mathbb{N}$ such that $F_1(x/n)$ is a true formula.
6. $F = \forall x F_1$ 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 $\hat{g}öd$:

1. $e(n) = (0, n)$ for all $n \in \mathbb{N}$.
2. $e(x_i) = (1, i)$ for all $i \in \mathbb{N}$.
3. $e(s + t) = (2, \langle e(s), e(t) \rangle)$ for all terms $s$ and $t$.
4. $e(s \cdot t) = (3, \langle e(s), e(t) \rangle)$ for all terms $s$ and $t$.
5. $e(s = t) = (4, \langle e(s), e(t) \rangle)$ for all terms $s$ and $t$.
6. $e(\neg F) = (5, e(F))$ for all formulas $F$.
7. $e(F \lor G) = (6, \langle e(F), e(G) \rangle)$ for all formulas $F$ and $G$.
8. $e(F \land G) = (7, \langle e(F), e(G) \rangle)$ for all formulas $F$ and $G$.
9. $e(\exists x_i F) = (8, \langle i, e(F) \rangle$ for all formulas $F$. 

© Markus Bläser 2007–2015
10. \( e(\forall x_i F) = \langle 9, (i, e(F)) \rangle \) for all formulas \( F \).

It is easy to see that the set \( \text{im} e \) is decidable. Like for \( \text{g"od} \), the concrete construction is not so important.

**Definition 11.4** The set of all encodings of true formulas is denoted by \( T \).

### 11.2 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 \( F \), 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 occurrence of \( y_\kappa \) is replaced by \( n_\kappa \), \( 1 \leq \kappa \leq k \).

**Definition 11.5** 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 \( n_1, \ldots, n_k, s \in \mathbb{N} \). In the same way, we define arithmetical representability for functions \( f : \mathbb{N}^k \to \mathbb{N}^m \).

**Example 11.6**

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

\[
((y_1 + z) = y_2) \lor ((y_1 > y_2) \land (z = 0))).
\]

Above, \( (y_1 > y_2) \) shorthands \( \exists h(y_1 = y_2 + h + 1) \).

3. Division with remainder is arithmetically representable, too: \( y_1 \text{ DIV } y_2 \) is represented by

\[
\exists r((r < y_2) \land (y_1 = z \cdot y_2 + r))
\]

and \( y_1 \text{ MOD } y_2 \) is represented by

\[
\exists q((r < y_2) \land (y_1 = q \cdot y_2 + r))
\]

\(^2\)If \( f(n_1, \ldots, n_k) \) is undefined, then \( F(n_1, \ldots, n_k, s) \) is not true for all \( s \in \mathbb{N} \).

© Markus Bläser 2007–2015
11.2.1 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 $\pi_1$ and $\pi_2$ of the pairing function $\langle \cdot , \cdot \rangle$. 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 11.7 (Chinese remainder theorem—concrete version)** Let $n_1, \ldots, n_t$ be pairwise coprime, i.e., $\gcd(n_i, n_j) = 1$ for $i \neq j$. Then the mapping

$$\pi_{n_1,\ldots,n_t} : \{0, \ldots, n_1 \cdot \ldots \cdot n_t - 1\} \rightarrow \{0, \ldots, n_1 - 1\} \times \cdots \times \{0, \ldots, n_t - 1\}$$

$$m \mapsto (m \mod n_1, \ldots, m \mod n_t)$$

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

$$c_1 n_1 \mod n_2 = 1 \quad \text{and} \quad c_2 n_2 \mod n_1 = 1.$$ 

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

$$a \mod n_1 = a_1 \quad \text{and} \quad a \mod n_2 = a_2$$

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 $a' = a + in_1 n_2$ is. Since $n_1, n_2 \mid in_1 n_2$,

$$a' \mod n_1 = a_1 \quad \text{and} \quad a' \mod n_2 = a_2,$$

too.

---

3 $i \mod j$ 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 to it right away.

5 We can get such integers, the so-called cofactors, via the extended Euclidian algorithm for computing gcds.
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 $(m \text{ mod } N) \text{ mod } n_i = i \text{ mod } n_i$ 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.

\[\square\]

Lemma 11.8 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. \[\square\]

Lemma 11.9 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 $a_\kappa$ for $x$.

Proof. Consider

$$M(x, u, v, w) = (x = v \mod (1 + uw)) \land (v < 1 + uw).$$

Now set $s = \max\{a_1, \ldots, a_k, k\}$. By Lemma 11.8, 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 definition. Thus the formula $M(a_\kappa, \kappa, A, S)$ is true. The second part of $M$ ensures that no other value fulfills $M(x, \kappa, A, S)$. \[\square\]

11.2.2 Main result

Theorem 11.10 Every WHILE computable function is arithmetically representable.

Proof. We show by structural induction that for every WHILE program $P$ then function $\Phi_P : \mathbb{N}^\ell \to \mathbb{N}^\ell$, where $\ell$ is the largest index of a variable in $P$, is arithmetically representable by a formula $F_P(y_0, \ldots, y_k, z_1, \ldots, z_k)$. 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

\[
F_P(y_0, \ldots, y_k, z_0, \ldots, z_k) = (z_i = y_j + y_k) \land \bigwedge_{m \neq i} z_m = y_m.
\]

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 11.6.

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 \( \Phi_{P_i} : \mathbb{N}^\ell \rightarrow \mathbb{N}^\ell \) are arithmetically representable by formulas \( F_{P_i} \), \( i = 1, 2 \). We have \( \Phi_P = \Phi_{P_2} \circ \Phi_{P_1} \). The formula

\[
F_P = \exists a_0 \ldots \exists a_\ell F_{P_1}(y_0, \ldots, y_\ell, a_0, \ldots, a_\ell) \land F_{P_2}(a_0, \ldots, a_\ell, z_0, \ldots, z_\ell)
\]

represents \( \Phi_P \); the variables \( a_0, \ldots, a_\ell \) “connect” the two formulas in such a way that the output of \( \Phi_{P_1} \) becomes the input of \( \Phi_{P_2} \).

It remains the case \( P = \text{while } x_i \neq 0 \text{ do } P_1 \text{ od} \). Let \( F_{P_1} \) be a formula that represents \( \Phi_{P_1} \). 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 11.9.

\[
F_P = \exists a_0 \exists S_0 \ldots \exists a_\ell \exists S_\ell \exists t
\]

\[
(M(y_0, 0, A_0, S_0) \land \ldots \land M(y_\ell, 0, A_\ell, S_\ell) \land
M(z_0, t, A_0, S_0) \land \ldots \land M(z_\ell, t, A_\ell, S_\ell) \land
\forall \tau \exists v((\tau \geq t) \lor (M(v, \tau, A_i, S_i) \land (v > 0))) \land
M(0, t, A_i, S_i) \land
\forall \tau \exists a_0 \ldots \exists a_\ell \exists b_0 \ldots \exists b_\ell
(F_{P_1}(a_0, \ldots, a_\ell, b_0, \ldots, b_\ell) \land
M(a_0, \tau, A_0, S_0) \land \ldots \land M(a_\ell, \tau, A_\ell, S_\ell) \land
M(b_0, \tau + 1, A_0, S_0) \land \ldots \land M(b_\ell, \tau + 1, A_\ell, S_\ell) \land
(\tau \geq t)))
\]

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 \( x_\lambda \) is \( y_\lambda \), \( 0 \leq \lambda \leq \ell \). The third line ensures that after the \( t \)th execution, the value of the variable \( x_\lambda \) is \( z_\lambda \), \( 0 \leq \lambda \leq \ell \). The fourth and fifth line ensures that the first time that \( x_i \) contains the value 0 is after the \( t \)th 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

\footnote{\(P_1 \) or \( P_2 \) might not contain the variable \( x_\ell \). We pad \( \Phi_{P_1} \) to a function \( \mathbb{N}^\ell \rightarrow \mathbb{N}^\ell \) in the obvious way.}
The values that we get if we run $P_1$ with $x_0, \ldots, x_\ell$ containing the values after $\tau$th execution. Note that the formula $M$ is satisfied by at most one value for fixed $\tau$, $A_\lambda$, and $S_\lambda$. This ensure consistency, i.e., even if we $A_\lambda$ and $S_\lambda$ do not contain the values from Lemma 11.9, if the formula $F_P$ is satisfied, then the values stored in $A_\lambda$ and $S_\lambda$, $0 \leq \lambda \leq \ell$, correspond to an execution of the WHILE program $P$. ■

Remark 11.11 Furthermore, there is a WHILE program that given $\text{göd}(P)$, computes the encoding of a formula presenting $\varphi_P$.

Lemma 11.12 If $\mathcal{T} \in \text{RE}$, then $\mathcal{T} \in \text{REC}$.

Proof. Let $f$ be a total WHILE computable functions such that $\text{im } f = \mathcal{T}$. A WHILE program $P$ that decides $\mathcal{T}$ first checks whether a given input $x \in \text{im } e$. 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 11.13 $\mathcal{T} \notin \text{RE}$.

Proof. Let $L \in \text{RE} \setminus \text{REC}$ and $F(y, z)$ be a formula that represents $\chi_L'$. 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 $\mathcal{T}$. ■

11.3 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 of finite size. Thus we can view them as natural numbers by using any “easy” injective mapping into $\mathbb{N}$.

Definition 11.14 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. $F : P \rightarrow L$ 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 11.15** A proof system $(P, F)$ for $L$ is complete if $F$ is surjective.

**Theorem 11.16** There is no complete proof system for the set of all true arithmetic formulas $T$.

**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 $\text{im } f = T$. But this contradicts Theorem 11.13. ■
12 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.

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.

12.1 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 $\Gamma$, the tape alphabet. (Here, finite alphabet is just a fancy word for finite set.) $\Gamma$ contains one distinguished symbol, the blank $\Box$. Each tape is two-sided infinite. That is, we can formally model it as a function $T : \mathbb{Z} \rightarrow \Gamma$ and $T(i)$ denotes the content of the $i$th 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,

---

1In some textbooks, the tapes are only one-sided infinite. As we will see soon, this does not make any difference.
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
\[
\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k.
\]

$\delta$ 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 $\gamma_1$ by $\gamma'_1$ on the first tape, $\gamma_2$ by $\gamma'_2$ 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 12.1 is a schematic of a Turing machine. I do not know whether it is really helpful.

**Definition 12.1** 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. \( \Sigma \) is a finite alphabet, the input alphabet.

3. \( \Gamma \) is a finite alphabet, the tape alphabet. There is a distinguished symbol \( \square \in \Gamma \), the blank. We have \( \Sigma \subseteq \Gamma \setminus \{\square\} \).

4. \( \delta : Q \times \Gamma^{k} \rightarrow Q \times \Gamma^{k} \times \{L,S,R\}^{k} \) is the transition function.

5. \( q_{0} \in Q \) is the start state.

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 \( \delta \).

**Example 12.2** Let us consider a first example: We will construct a 1-tape Turing machine \( \text{INC} \) 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. What does \( \text{INC} \) do? If the lowest order bit is a 0, then \( \text{INC} \) replaces it by a 1 and is done. If the lowest order bit is 1, then \( \text{INC} \) replaces it by a 0. This creates a carry and \( \text{INC} \) 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.

\( \text{INC} \) has three states \( \text{add} \), \( \text{back} \), and \( \text{stop} \). The state \( \text{add} \) 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:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>( \square )</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>(back, 1, L)</td>
<td>(add, 0, R)</td>
<td>(back, 1, L)</td>
</tr>
<tr>
<td>back</td>
<td>(back, 0, L)</td>
<td>(back, 1, L)</td>
<td>(stop, ( \square ), R)</td>
</tr>
<tr>
<td>stop</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Above, “—” stands for undefined. In the state \( \text{add} \), \( \text{INC} \) goes to the right replacing every 1 by a 0 until it finds the first 0 or \( \square \). This 0 or \( \square \) is then replaced by a 1 and \( \text{INC} \) enters the state \( \text{back} \). In the state \( \text{back} \), \( \text{INC} \) goes to the left leaving the content of the cells unchanged until it finds the first \( \square \). It goes one step to the right and is done.

Instead of a table, a transition diagram is often more understandable. Figure 12.2 show this diagram for the Turing machine of this exercise. 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 $\alpha$, then it goes to state $q'$, writes $\beta$, and moves its head as given by $r \in \{L, S, R\}$.

12.2 Configurations and computations

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 □. The only exception is the first tape on which the input stands in unvisited cells at the beginning. So instead of modeling a tape as a function $\mathbb{Z} \rightarrow \Gamma$ (which is the same as a two-sided infinite word over $\Gamma$), 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 $(q, (p_1, x_1), \ldots, (p_k, x_k)) \in Q \times (\mathbb{N} \times \Gamma^*)^k$ such that $1 \leq p_\kappa \leq |x_\kappa|$ for all $1 \leq \kappa \leq k$. $q \in Q$ is the current state of $M$. $x_1, \ldots, x_k$ is the content of the cells visited so far of the tapes $1, \ldots, k$. $p_\kappa$ denotes the position of the head of the tape $\kappa$, $1 \leq \kappa \leq k$. We store the position relatively, i.e., $p_\kappa$ denotes the position within $x_\kappa$ 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 $(q_0, (1, w), (1, \square), \ldots, (1, \square))$. 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 □. We usually denote the start configuration by $SC_M(w)$.

Let $C = (q, (p_1, x_1), \ldots, (p_k, x_k))$ and $C' = (q', (p'_1, x'_1), \ldots, (p'_k, x'_k))$ 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, $\alpha_\kappa$ 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
12.2. Configurations and computations

$M$. Formally this means that if $\delta(q, \alpha_1, \ldots, \alpha_k) = \langle q', \beta_1, \ldots, \beta_k, r_1, \ldots, r_k \rangle$, 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 $x_\kappa$ 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 $\ell$ such that $C \vdash_M C_1 \vdash_M \cdots \vdash_M C_\ell \vdash_M C'$. 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 $SC_M(w) \vdash^*_M C_t$ 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 $SC_M(w) \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_t$ a computation of $M$ on $w$. If $M$ does not halt on $w$, then the corresponding computation is infinite.

Assume that $SC_M(w) \vdash^*_M C_t$ and $C_t = \langle q, (p_1, x_1), \ldots, (p_k, x_k) \rangle$ is a halting configuration. Let $i \leq p_1$ be the largest index such that $x_1(i) = \square$. 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 $x_1(j) = \square$. 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 $w$. 

© Markus Bläser 2007–2015
12.3 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 12.3** 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 $\Sigma$. So we either have to put 0 and 1 into $\Sigma$ or we have to identify two symbols of $\Sigma$ with 0 and 1. While this works, there is a more elegant way (and this is the one you usually will find in books, too): A Turing machine is now described by a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$. $Q_{\text{acc}} \subseteq Q$ 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 12.4** 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 $M$.

Thus halting in an accepting configuration means outputting 1 and halting in a rejecting configuration means outputting 0.

---

²So 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.
13 Examples, tricks, and syntactic sugar

In the beginning, we will do low level descriptions of the Turing machine, that is, we will write down the transition functions explicitly.

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.

13.1 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 13.1 erases the cells to the right of the head until the first blank is found.

The machine COPY in Figure 13.2 copies 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 13.3 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

![Diagram of Turing machine ERASE](image13.1.png)

Figure 13.1: The Turing machine ERASE
the state whether the content is zero or not. The Turing machine stops in
the state yes or no.

**Exercise 13.1** Construct a Turing machine DEC that decreases the content
of the tape by 1 if the content is $> 0$.

### 13.2 Some techniques and tricks

#### 13.2.1 Concatenation

Although Turing machine 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 13.1. 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

---

© Markus Bläser 2007–2015
13.2. Some techniques and tricks

Figure 13.4: 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 13.5: 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.

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 13.5.

13.2.2 Loops
13.2.3 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 □. 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 $\gamma$ 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 $\Gamma$. It then replaces this symbol $\bar{\gamma}$ by $\gamma$ and continues its computation. (Leaving more than one marker in this way per tape could mean trouble!)

13.2.4 Storing information in the state

Have another look at the Turing machine COMPARE. After it reached the first 1 or the first □ 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 □. 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 □”. 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 $\{\text{back}\} \times \{1, \Box\}$ as states.

13.2.5 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 $N$,

$$\Delta : (Q \times Q') \times \Gamma^{k+k'} \rightarrow (Q \times Q') \times \Gamma^{k+k'} \times \{L, S, R\}^{k+k'},$$

is defined by

$$\Delta((q, q'), \gamma_1, \ldots, \gamma_{k+k'}) = ((p, p'), \alpha_1, \ldots, \alpha_k, \alpha_{k+1}', \ldots, \alpha_{k'}', r_1, \ldots, r_k, r_1', \ldots, r_{k'}')$$

if

$$\delta(q, \gamma_1, \ldots, \gamma_k) = (p, \alpha_1, \ldots, \alpha_k, r_1, \ldots, r_k)$$

and

$$\delta(q', \gamma_{k+1}', \ldots, \gamma_{k+k'}) = (p', \alpha_1', \ldots, \alpha_k', \alpha_{k+1}', \ldots, \alpha_{k'}', r_1', \ldots, r_{k'})$$

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 \textsc{Copy}, before \( N \) starts with the simulation).

Here is one application: take \( M \) to be any machine and \( M' \) is the machine from Figure 13.5. Modify the function \( \Delta \) 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.

13.3 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 \textsc{INC}:

\begin{itemize}
  \item \textbf{Input:} \( x \in \{0,1\}^* \), viewed as a number in binary.
  \item \textbf{Output:} \( x \) increased by 1
  \begin{enumerate}
    \item Go to the right and replace every 1 by a 0 until you reach the first 0 or \( \square \).
    \item Write a 1 and go back to the right to the beginning of the string.
  \end{enumerate}
\end{itemize}

Once we got even more experienced with Turing machines, we could even write:

\begin{itemize}
  \item \textbf{Input:} \( x \in \{0,1\}^* \), viewed as a number in binary.
  \item \textbf{Output:} \( x \) increased by 1
  \begin{enumerate}
    \item Increase the content of the tape by 1.
  \end{enumerate}
\end{itemize}
Warning!!! Although the example above suggests it, the sentence “The Turing machine produces the desired output” 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.)

13.4 Further exercises

Exercise 13.2 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!
14 Church–Turing thesis

14.1 WHILE versus Turing computability

In this chapter, we want to show that

\[
\text{WHILE computable} \quad \text{equals} \quad \text{Turing computable.}
\]

But there is of course a problem. WHILE programs compute functions \( \mathbb{N}^* \to \mathbb{N} \) 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 \( \Sigma = \{0, 1\} \).

14.1.1 \( \mathbb{N} \) 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 \( \varepsilon \) 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, but 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 \( \varepsilon \). 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 \( \text{cod} \). More compactly, we can write \( \text{cod}(x) = \text{bin}^{-1}(1x) - 1 \). (Note that we can write \( \text{bin}^{-1} \), since \( 1x \) is in the image of bin.)
**Exercise 14.1** Show that \( \text{cod} \) is indeed bijective.

**14.1.2 \( \mathbb{N} \to \mathbb{N} \) 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:

\[
\hat{f}(x) = \text{cod}^{-1}(f(\text{cod}(x))) \quad \text{for all } x \in \{0,1\}^*.
\]

Or, in other words, the following diagram commutes:

\[
\begin{array}{ccc}
\mathbb{N} & \xrightarrow{f} & \mathbb{N} \\
\downarrow \text{cod}^{-1} & & \downarrow \text{cod}^{-1} \\
\{0,1\}^* & \xrightarrow{\hat{f}} & \{0,1\}^*
\end{array}
\]

Conversely, if \( g : \{0,1\}^* \to \{0,1\}^* \), then \( \hat{g} : \mathbb{N} \to \mathbb{N} \) is defined by

\[
\hat{g}(n) = \text{cod}(g(\text{cod}^{-1}(n))) \quad \text{for all } n \in \mathbb{N}.
\]

In other words,

\[
\begin{array}{ccc}
\{0,1\}^* & \xrightarrow{g} & \{0,1\}^* \\
\downarrow \text{cod} & & \downarrow \text{cod} \\
\mathbb{N} & \xrightarrow{\hat{g}} & \mathbb{N}
\end{array}
\]

**Exercise 14.2** 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 14.1** The mapping \( \text{cod} \) is not too natural. For instance, with \( f : \mathbb{N} \to \mathbb{N} \), we could associate the mapping \( \text{bin}(x) \mapsto \text{bin}(f(x)) \). But if \( \text{cod} \) is a bijection, then we have the nice property that \( \hat{\cdot} \) is self-inverse.

**Exercise 14.3** Show that \( \text{cod} \) and \( \text{cod}^{-1} \) 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 \( \text{cod}^{-1}(n) \) and stores them in an array.

2. Construct\(^1\) a Turing machine that given \( x \in \{0,1\}^* \), writes \( \text{cod}(x) \) many 1’s on the first tape.

So the only reason why a WHILE program or Turing machine cannot compute \( \text{cod} \) or \( \text{cod}^{-1} \) 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.
14.1.3 Pairing functions

We also need a pairing function for \{0, 1\}*, i.e., an injective mapping \(\langle.,.\rangle : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}^*\). If we take our pairing function \(\langle.,.\rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\), then of course,

\[(x, y) \mapsto \text{cod}^{-1}(\langle\text{cod}(x), \text{cod}(y)\rangle)\]

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 \(\beta(x) = x_10x_20\ldots x_{|x|}0\), 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 \(\beta(\text{bin}(|x|))11xy\). By scanning for the first pair 11, we can divide the string into \(\beta(\text{bin}(|x|))\) 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 14.4** Try to get even shorter encodings in this way. What is the shortest that you can get?

14.2 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 \(s_\mu\) 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. \(\text{if } x_i \neq 0 \text{ then goto } \lambda\)

The semantics of the first three statements is the same as for WHILE programs. After the \(\mu\)th statement is executed, the program goes on with the \((\mu+1)\)th statement. The only exception is the *conditional jump* if \(x_i \neq 0\) then goto \(\lambda\). If the content of \(x_i\) is zero, then we go on with the \((\mu + 1)\)th statement, otherwise, we go on with statement \(\lambda\). 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—\( \varphi_P \).

**Exercise 14.5** Show how to simulate an unconditional jump in GOTO. (An unconditional jump, we denote it by \texttt{goto} \( \lambda \), always jumps to line \( \lambda \) no matter what.)

**Exercise 14.6** 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 \( \Phi_P \) 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
  \( P \)
end
```

is simulated by

1: if \( x_i \neq 0 \) then goto 3
2: goto 5
3: \( P \)
4: goto 1
5: \dots

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 \texttt{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 14.2** For every WHILE program \( P \) there is a GOTO program \( Q \) with \( \varphi_P = \varphi_Q \).

### 14.3 Turing machines can simulate GOTO programs

**Lemma 14.3** 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 $\text{cod}^{-1}(n)$. For the simulation, it is easier to have $\text{bin}(n)$ on the tape. We get it by appending a 1 to $\text{cod}^{-1}(n)$ and then subtracting 1 by using the Turing machine DEC.

We will use the Turing machines INC, DEC, ERASE, and COMPARE (see Section 13) to construct a Turing machine $M$ that simulates $P$, more precisely, computes $\hat{f}$.

For each instruction $(\mu, s_\mu)$, we have a state $q_\mu$. 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 $s_\mu$ and all heads stand on the left-most symbol that is not a blank. (The lowest order bit is standing on the left.)

Figure 14.1 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 $s_\mu$, $M$ goes from $q_\mu$ 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 $\lambda$. 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 $(\mu, V)$ and that $\Phi_P(\mu, V) = (\mu', V')$. If $M$ is in state $q_\mu$, the content of the tapes are $\text{bin}(V(\lambda))$, $0 \leq \lambda \leq \ell$, and the heads are standing on the lowest order bits of $\text{bin}(V(\lambda))$, 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 $\text{bin}(V'(\lambda))$, $1 \leq \lambda \leq \ell$, and the heads are standing on the lowest order bits of $\text{bin}(V(\lambda))$.

From this claim, the correctness of the simulation follows immediately. When $M$ stops, we have to transform the binary expansion $b = \text{bin}(n)$ on tape 1 back into $\text{cod}^{-1}(n)$, which is easy. ■

Exercise 14.7 Give a detailed description of the general construction.
Exercise 14.8 Prove the claim in the proof of Lemma 14.3

14.4 WHILE programs can simulate Turing machines

Lemma 14.4 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 \( \Gamma \) by the numbers \( 0,1,\ldots,s-1 \). We Exercise 13.2, we can assume that each tape is one-sided infinite.

The content of each tape \( \kappa \) is stored in an array \( A_\kappa \). \( A_\kappa[i] = j \) means that the cell \( i \) contains the symbol that corresponds to \( j \). Of course, \( A_\kappa \) always stores only a finite amount of data. Recall that the arrays that we created in WHILE are dynamic, so we can extend them whenever \( M \) visits a new cell. The variable \( p_\kappa \) contains the (absolute) position of the head.

In the beginning, we have to write \( \text{cod}^{-1}(x_0) \) into \( A_1 \). We can do this using the Turing machine constructed in Exercise 14.3.

The domain of \( \delta \) is finite, thus we can hardwire the table of \( \delta \) 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

\[
\text{while } \delta(q, a_1, \ldots, a_k) \text{ is defined do } \ldots \text{ od}
\]
From the construction, it is clear that the simulation is correct. ■

**Corollary 14.5 (Kleene normal form)** For every WHILE computable function \( f \), there are FOR programs \( P_1, P_2, P_3 \) and a WHILE program of the form

\[
P_1; \text{while } x_i \neq 0 \text{ do } P_2 \text{ od}; P_3
\]

that computes \( f \).

**Proof.** 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. ■

### 14.5 Church–Turing thesis

From Lemmas 14.2, 14.3, and 14.4, we get the following result.

**Theorem 14.6** 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.
B Primitive and \( \mu \)-recursion

Historically, primitive recursive functions and \( \mu \)-recursive functions are one of the first concepts to capture “computability”.

\section*{B.1 Primitive recursion}

We are considering functions \( \mathbb{N}^s \rightarrow \mathbb{N} \) for any \( s \geq 1 \).

\begin{definition}
The set of all primitive recursive functions is defined inductively as follows:
\begin{enumerate}
\item Every constant function is primitive recursive.
\item Every projection \( p^s_i : \mathbb{N}^s \rightarrow \mathbb{N} \) (mapping \( (a_1, \ldots, a_s) \) to \( a_i \)) is primitive recursive.
\item The successor function \( \text{suc} : \mathbb{N} \rightarrow \mathbb{N} \) defined by \( \text{suc}(n) = n + 1 \) is primitive recursive.
\item If \( f : \mathbb{N}^s \rightarrow \mathbb{N} \) and \( g_i : \mathbb{N}^t \rightarrow \mathbb{N}, 1 \leq i \leq s \), 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. This scheme is called primitive recursion.
\item If \( g : \mathbb{N}^s \rightarrow \mathbb{N} \) and \( h : \mathbb{N}^{s+2} \rightarrow \mathbb{N} \) are primitive recursive, then the function \( f : \mathbb{N}^{s+1} \rightarrow \mathbb{N} \) defined by
\[ f(0, a_1, \ldots, a_s) = g(a_1, \ldots, a_s) \]
\[ f(n + 1, a_1, \ldots, a_s) = h(f(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s) \]
is primitive recursive. This scheme is called primitive recursion.
\end{enumerate}
\end{definition}

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:

\begin{example}
The function \( \text{add}(x, y) = x + y \) is primitive recursive. We have
\begin{align*}
\text{add}(0, y) &= y, \\
\text{add}(x + 1, y) &= \text{suc}(\text{add}(x, y)).
\end{align*}
\end{example}
Above, we did not write down the projections explicitly. The correct definition looks like this:

\[
\begin{align*}
\text{add}(0, y) &= p_1^1(y), \\
\text{add}(x + 1, y) &= \text{suc}(p_1^3(\text{add}(x, y), x, y)).
\end{align*}
\]

Since this looks somewhat confusing, we will omit the projections in the following.

**Exercise B.1** Show that the function \(\text{mult}(x, y) = xy\) is primitive recursive.

**Example B.3** The predecessor function defined by

\[
\text{pred}(n) = \begin{cases} 
  n - 1 & \text{if } n > 0 \\
  0 & \text{otherwise}
\end{cases}
\]

is primitive recursive. We have

\[
\begin{align*}
\text{pred}(0) &= 0, \\
\text{pred}(n + 1) &= n.
\end{align*}
\]

This is a primitive recursion scheme.

**Exercise B.2** Prove that the modified difference \(\text{sub}(x, y) = \max\{x - y, 0\}\) is primitive recursive.

### B.1.1 Bounded maximization

Finally, we will show that the pairing function \(\langle \ldots \rangle\) and “its inverses” \(\pi_1\) and \(\pi_2\) 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

\[
\text{bounded-max}-P(n) := \max\{x \leq n \mid P(x) = 1\}
\]

is primitive recursive. (If no such \(x\) exists, i.e., the maximum is undefined, we set \(\text{bounded-max}-P(n) = 0\).) This can be seen as follows:

\[
\begin{align*}
\text{bounded-max}-P(0) &= 0 \\
\text{bounded-max}-P(n + 1) &= \begin{cases} 
  n + 1 & \text{if } P(n + 1) = 1 \\
  \text{bounded-max}-P(n) & \text{otherwise}
\end{cases} \\
&= (1 - P(n + 1)) \cdot \text{bounded-max}-P(n) + P(n + 1) \cdot (n + 1).
\end{align*}
\]
In the same way, we can see that the *bounded existential quantifier* defined by

\[ \text{bounded-}\exists P(n) := \begin{cases} 1 & \text{if there is an } x \leq n \text{ with } P(x) = 1 \\ 0 & \text{otherwise} \end{cases} \]

is primitive recursive:

\[ \text{bounded-}\exists P(0) = P(0) \]
\[ \text{bounded-}\exists P(n + 1) = P(n + 1) + \text{bounded-}\exists P(n)(1 - P(n + 1)) \]

Above, \( P \) has only one argument. It is easy to see that for a predicate with \( s \) arguments,

\[ \text{bounded-max}_i P(x_1, \ldots, x_s) := \max \{ x \leq x_i \mid P(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_s) = 1 \} \]

is primitive recursive. In the same way, we can define \( \text{bounded-}\exists_i P \) 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 can write:

\[ \pi_1(z) = \text{bounded-max}_1 Q(x, \text{bounded-}\exists_2 Q(x, y, z), z) \]
\[ \pi_2(z) = \text{bounded-max}_2 Q(\text{bounded-}\exists_1 Q(x, y, z), y, z) \]

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.

### B.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*

\[ f_i(0, a_1, \ldots, a_s) = g_i(a_1, \ldots, a_s), \]
\[ f_i(n + 1, a_1, \ldots, a_s) = h_i(f_1(n, a_1, \ldots, a_s), \ldots, f_t(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s), \]

for \( i = 1, \ldots, t, \) are primitive recursive. To see this, define \( f \) by

\[ f(0, a) = \langle g_1(\pi_1(a), \ldots, \pi_s(a)), \ldots, g_t(\pi_1(a), \ldots, \pi_s(a)) \rangle \]
\[ f(n + 1, a) = \langle h_1(\pi_1(f(n, \pi_1(a), \ldots, \pi_s(a))), \ldots, \pi_1(f(n, \pi_1(a), \ldots, \pi_s(a))), n, \pi_1(a), \ldots, \pi_s(a)), \ldots, h_t(\pi_1(f(n, \pi_1(a), \ldots, \pi_s(a))), \ldots, \pi_1(f(n, \pi_1(a), \ldots, \pi_s(a))), n, \pi_1(a), \ldots, \pi_s(a)) \rangle. \]

\( f \) 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
\[ f_i(n, a_1, \ldots, a_s) = \pi_i(f(n, (a_1, \ldots, a_s))). \]

Thus each \( f_i \) is primitive recursive.

**B.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 B.4** Let \( P \) be a FOR program with \( s \) inputs. Let \( \ell \) 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 \( v_\lambda \) is the projection on the \( \lambda \)th 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
\[
(v_{i,0}(a_0, \ldots, a_\ell), \ldots, v_{i,\ell}(a_0, \ldots, a_\ell)) = \Phi_{P_i}(a_0, \ldots, a_\ell), \quad i = 1, 2.
\]

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 \( 0 \leq \lambda \leq \ell \) and \( a_0, \ldots, a_\ell \in \mathbb{N} \). Thus \( v_0, \ldots, v_\ell \) are primitive recursive.

If \( P = \text{for } x_i \text{ do } P_1 \text{ od} \), then by the induction hypothesis, there are functions \( v_{1,0}, \ldots, v_{1,\ell} \) such that
\[
(v_{1,0}(a_0, \ldots, a_\ell), \ldots, v_{1,\ell}(a_0, \ldots, a_\ell)) = \Phi_{P_1}(a_0, \ldots, a_\ell). \quad (B.1)
\]

Define \( u_0, \ldots, u_\ell \) by

\[ u_\lambda(0, a_0, \ldots, a_\ell) = a_\lambda \]
\[ u_\lambda(n + 1, a_0, \ldots, a_\ell) = v_{1,\lambda}(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_0, \ldots, a_\ell)) \]
for \( 0 \leq \lambda \leq \ell \). This is a simultaneous primitive recursion scheme. We claim that
\[
(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_0, \ldots, a_\ell)) = \Phi_{P_2}^{(n)}(a_0, \ldots, a_\ell) \quad (B.2)
\]

© Markus Bläser 2007–2015
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

\[
\begin{align*}
u_\lambda(n + 1, a_0, \ldots, a_\ell) &= v_{1, \lambda}(u_0(n, a_0, \ldots, a_\ell), \ldots, u_\ell(n, a_1, \ldots, a_\ell)) \\
&= v_{1, \lambda}(\Phi P_1^{(n+1)}(a_0, \ldots, a_\ell)) \\
&= \lambda\text{th entry of } \Phi P_1^{(n+1)}(a_0, \ldots, a_\ell).
\end{align*}
\]

The last equality follows from the induction hypothesis (B.1).

Altogether, this shows (B.2). We get \( v_\lambda \) by \( v_\lambda(a_0, \ldots, a_n) = u_\lambda(a_i, a_0, \ldots, a_n) \).

\[\square\]

**Lemma B.5** For every primitive recursive function \( f \), there is a FOR program \( P \) with \( \phi_P = f \).

**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

\[
\begin{align*}
f(0, a_1, \ldots, a_s) &= g(a_1, \ldots, a_s), \\
f(n + 1, a_1, \ldots, a_s) &= h(f(n, a_1, \ldots, a_s), n, a_1, \ldots, a_s),
\end{align*}
\]

then there are programs \( P \) and \( Q \) that compute \( h \) and \( g \), respectively. Now the following program computes \( f(a_0, a_1, \ldots, a_s) \):

\[
\begin{align*}
1: & \quad x_0 := g(a_1, \ldots, a_s); \\
2: & \quad \textbf{for } a_0 \textbf{ do} \\
3: & \quad x_0 := h(x_0, a_0, a_1, \ldots, a_s) \\
4: & \quad \textbf{od}
\end{align*}
\]

We saw how to simulate function calls of FOR computable functions. \( \square \)

**Theorem B.6** A function \( f \) is primitive recursive iff it is FOR computable.

**Proof.** The “\( \Leftarrow \)”-direction is Lemma B.5. The function \( v_0 \) of Lemma B.4 is the function that is computed by the program \( P \). This show the other direction. \( \square \)
B.2 $\mu$-recursion

The $\mu$-operator allows unbounded search.

**Definition B.7** Let $f : \mathbb{N}^{s+1} \to \mathbb{N}$. The function $\mu f : \mathbb{N}^s \to \mathbb{N}$ is defined by

$$\mu f(a_1, \ldots, a_s) = \min \{ n \mid f(n, a_1, \ldots, a_s) = 0 \text{ and } \forall m < n, f(m, a_1, \ldots, a_s) \text{ is defined} \}.$$  

**Definition B.8** The set of all $\mu$-recursive functions is defined inductively as in Definition B.1 except that the set is closed under $\mu$-recursion instead of primitive recursion.

**Theorem B.9** A function is $\mu$-recursive iff it is WHILE computable.

**Proof.** This proof is just an “add-on” to the proof of Theorem B.6.

For the “$\Rightarrow$”-direction, we just have to consider one more case in the proof of Lemma B.5. If $f = \mu g$ for some $\mu$-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: \hspace{1em} $n := n + 1$
4: \hspace{1em} 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 $P = \text{while } x_i \neq 0 \text{ do } P_1 \text{ od}$. The functions $u_\lambda(n, a_0, \ldots, a_\ell)$ constructed in the proof of Lemma B.4 is the content of the variable $x_\lambda$ 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 $x_\lambda$ after executing $P$. ■

---

**Excursus: Programming systems III**

We can also assign Gödel numbers to recursions schemes. We start by assigned numbers to the constant functions, projections, and successor function. For instance, $(0, (s, c))$ could stand for the function of arity $s$ that has the value $c$.  

© Markus Bläser 2007–2015
everywhere. \( \langle 1, (s, i) \rangle \) encodes \( p_s^i \) and so on. Then we define Gödel numbers for composition and primitive and \( \mu \)-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 \( \theta_i \) be the function that is computed by the recursion scheme with Gödel number \( i \). If \( i \) is not a valid Gödel number, then \( \theta_i \) 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.

Once we have constructed a universal Turing machine and/or WHILE program, we will see that it is also universal. It is clearly acceptable, since composition is directly available in recursion schemes.
15 A universal Turing machine

15.1 Gödel numberings for Turing machines

We want to find a mapping that maps Turing machines to strings in \( \{0, 1\}^* \). Let \( M = (Q, \{0, 1\}, \Gamma, \delta, q_0, Q_{acc}) \) be a \( k \)-tape Turing machine. We can assume that \( Q = \{0, 1, \ldots, s\} \) and \( \Gamma = \{0, 1, \ldots, \ell\} \). We assume that \( \ell \) is the blank.

We can encode a state \( q \) by \( \text{bin}(q) \) and a symbol \( \gamma \) 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
\[
\hat{r}_k = \begin{cases} 
00 & \text{if } r_k = S \\
10 & \text{if } r_k = L \\
01 & \text{if } r_k = R 
\end{cases}
\]

[... ] denotes (one of) the pairing functions discussed in Section 14.1.3. 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 \( \text{göd}_{TM} \) 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 \( \Gamma \),
- the encodings of \( \delta(q, \gamma_1, \ldots, \gamma_k) \), \( q \in Q \), and \( \gamma_1, \ldots, \gamma_k \in \Gamma \), in lexicographic order,
• \(\text{bin}(q_0)\), 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 \(\text{göd}_TM\) is an injective mapping.

**Definition 15.1** A mapping \(g\) from the set of all Turing machines over the input alphabet \(\{0, 1\}^*\) to \(\{0, 1\}^*\) is called a Gödel numbering if

1. \(g\) is injective,
2. there is a Turing machine \(C\) that computes the characteristic function of \(\text{im } g\), and
3. there is a Turing machine \(U\) that given \(i \in \text{im } g\) and \(x \in \{0, 1\}^*\) computes \(\varphi_M(x)\) where \(M = g^{-1}(i)\).

Constructing the Turing machine \(C_{TM}\) that computes the characteristic function of \(\text{im } \text{göd}_TM\) is rather easy. A \(i \in \text{im } \text{göd}_TM\) is just a concatenation of some tuples, we just have to check whether they have the right form.

### 15.2 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 it 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 \(i\)th cells of the \(k\) tapes of \(M\) contain the symbols \(i_1, \ldots, i_k\), then the \(i\)th block is

\[
\# \text{bin}(i_1)\# \text{bin}(i_2)\# \ldots \# \text{bin}(i_k).
\]

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

\[
\# \text{bin}(x_1)\# \text{bin}(\ell)\# \ldots \# \text{bin}(\ell)
\]
Input: \( i \in \{0,1\}^* \)
Output: accept, if \( i \in \text{im } \text{göd}_{TM} \), 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 tuples 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 15.1: The Turing machine \( C_{TM} \).

for \( j = 1, \ldots, n \) on it where \( x = x_1 x_2 \ldots x_n \) denotes the input for \( M \). Recall the \( \ell \) 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 \( j \)th tape of \( M \) is standing on the \( i \)th cell by replacing the \( j \)th \# of the \( i \)th 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 15.2** There is a Turing machine \( U_{TM} \) that, given a pair \([g,x]\) with \( g \in \text{im } \text{göd}_{TM} \) and \( x \in \{0,1\}^* \), computes \( \varphi_{göd_{TM}}^{-1}(g)(x) \).

**Exercise 15.1** Show that the constructed Turing machine \( U_{TM} \) is correct.
Input: \([g, x]\) with \(g\in\text{im}\, \varphi_{\text{TM}}^{-1}\) and \(x\in\{0,1\}^*\)
Output: \(\varphi_{\text{TM}}^{-1}(g)(x)\)

1. \(U_{\text{TM}}\) copies the start state to the third tape.
2. \(U_{\text{TM}}\) copies the input \(x\) from the first tape to the second tape as described above.
   It replaces all \# of the first block by \(*\).
3. \(U_{\text{TM}}\) moves the head to the first symbol of the leftmost block.
4. While the transition function is not undefined
   (a) \(U_{\text{TM}}\) goes to the right.
   (b) Whenever it finds a *, it copies the following number in binary to the fourth tape.
   (c) If \(U_{\text{TM}}\) reaches the right end of tape 2, then it looks up the tuple of \(\delta\) that corresponds to the current state (on tape 3) and the symbols copied to tape 4.
   (d) \(U_{\text{TM}}\) replaces the state on tape 3 by the new state.
   (e) \(U_{\text{TM}}\) goes to the left
   (f) Whenever \(U_{\text{TM}}\) finds a *, it updates the corresponding cells and moves the * to its new position.
5. If \(M := \varphi_{\text{TM}}^{-1}(g)\) is supposed to compute a function, then \(U_{\text{TM}}\) 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 15.2: The universal Turing machine \(U_{\text{TM}}\)
Kolmogorov Complexity

Kolmogorov complexity measures the “information” stored in a string \( x \in \{0,1\}^* \). While the string \( 0^n \) intuitively has low information content, since it can be described by “a sequence of \( n \) zeros” and we need \( \log n \) bits to write \( n \) down, the string \( 0110010010001111010110110010101 \) does not seem to have a short description. Komogorov complexity, like the halting problem, is another natural problem that is not Turing computable.

C.1 Definition

**Definition C.1** The Kolmogorov complexity \( K(x) \) of a string \( x \in \{0,1\}^* \) is the length of a shortest string \( y \in \{0,1\}^* \) such that the universal Turing machine \( U_{TM} \) outputs \( x \) on input \( y \).

This means, that we measure the number of bits that we need to produce \( x \). This can be seen as the ultimate compression task. The input \( y \) for \( U_{TM} \) is the compressed version of \( x \).

Why not just take the length of a shortest encoding of a Turing machine that outputs \( x \) on the empty word? The problem is that the encoding of a Turing machine that outputs \( x \) on the empty word usually needs \( |x| + 1 \) states. Thus the trivial encoding of \( x \) would have length \( \Theta(|x| \log |x|) \), which is not a disaster but also not very nice.

**Lemma C.2** There is a constant \( u \) such that \( K(x) \leq |x| + u \) for all \( x \in \{0,1\}^* \).

**Proof.** Let \( e \) be the Gödel number of a Turing machine that computes the identity. The universal Turing machine outputs \( x \) on input \([e, x]\). We have \( |[e, x]| \leq 2|e| + 2 + |x| \) (or \( \leq |e| + |x| + 2 \log |e| + 2 \) depending on our pairing function).\(^1\) Setting \( u = 2|e| + 2 \) concludes the proof. \( \blacksquare \)

**Example C.3** Consider the sequence \( z_n = 0^n \), \( n \in \mathbb{N} \). The following Turing machine outputs \( 0^n \) on input \( \text{bin}(n) \):

\(^1\)Here is the place where we need that the length of a pair \([a, b]\) can be bounded nicely in terms of \(|a|\) and \(|b|\).
Input: \( x = \text{bin}(n) \)

1. Count in binary to \( x \).
2. Every time you increase the counter, output “0”.

Thus \( K(z_n) \leq |\text{bin}(n)| + O(1) = \log |z_n| + O(1) \).

Exercise C.1 Let \( f : \mathbb{N} \to \mathbb{N} \) be total, injective, and WHILE computable. Let \( z_n = 0^{f(n)} \). Show that \( K(z_n) \leq \log f^{-1}(|z_n|) + O(1) \).

The exercise above shows that there are sequences of words whose Kolmogorov complexity grows very slowly. Are there sequences whose complexity is close to the upper bound of Lemma C.2?

Pitfall

Kolmogorov complexity only measures the length of the input string \( y \) for \( U_{TM} \). It does not say anything about the number of cells used by \( U_{TM} \) on input \( y \)!

Lemma C.4 For every natural number \( n \geq 1 \), there is a word \( w_n \in \{0,1\}^n \) such that \( K(w_n) \geq n \).

Proof. There are \( 1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1 \) strings in \( \{0,1\}^* \) of length \( < n \). Thus there are at most \( 2^n - 1 \) string with Kologorov complexity \( < n \), namely, the outputs of \( U_{TM} \) on these \( 2^n - 1 \) strings. There are \( 2^n \) words in \( \{0,1\}^n \). Thus at least one of them has Kolmogorov complexity \( n \).

Exercise C.2 Show that there are \( 2^n(1 - 2^{-i}) \) words \( w \) in \( \{0,1\}^n \) such that \( K(w) \geq n - i \).

We have defined Kolmogorov in terms of Turing machines. The definition of \( K \) depends on our universal Turing machine \( U_{TM} \). But there are many universal Turing machines. Does the choice of the Turing machine affect the Kolmogorov complexity? The answer is “not very much”. For a universal Turing machine \( V \), let \( K_V \) be the Kolmogorov complexity with respect to \( V \). In particular, the \( K \) that we used so far is \( K_{U_{TM}} \).

Lemma C.5 Let \( V \) and \( V' \) be universal Turing machines. Then there is a constant \( c_{V'} \) such that \( K_V(x) \leq K_V'(x) + c_{V'} \) for all \( x \in \{0,1\}^* \).
Proof. Let $y$ be a string of length $K_{V'}(x)$ such that $V'$ on input $y$ outputs $x$. Let $g' := \text{göd}_{TM}(V')$. Then $V$ outputs $x$ on input $[g', y]$. Thus

$$K_V(x) \leq |[g', y]| \leq 2|g'| + |y| = 2|g'| + 2 + K_{V'}(x).$$

Setting $c_{V'} = 2|g'| + 2$ concludes the proof. $\blacksquare$

Corollary C.6 For all universal Turing machines $V$ and $V'$, there is a constant $c_{V,V'}$ such that $|K_V(x) - K_{V'}(x)| \leq c_{V,V'}$ for all $x \in \{0,1\}^*$. 

Remark C.7 Above, the Turing machines $V$ and $V'$ could even work with different Gödel numberings. We could even compare Turing machines with any other acceptable programming system.

C.2 Kolmogorov random strings

Definition C.8 A string $x \in \{0,1\}^*$ is Kolmogorov random if $K(x) \geq |x|$. 

Kolmogorov random means that a string is incompressible. It does not have any “regular structure” that we could exploit to represent it by some shorter string. By Lemma C.4, Kolmogorov random strings exist. Kolmogorov randomness only speaks about one string at a time. It does not say anything about drawing strings at random. If we draw strings from $\{0,1\}^n$ uniformly at random, then we see every string with the same probability $2^{-n}$, in particular the string $0^n$ and any Kolmogorov random string, too.

Theorem C.9 Let $A$ be a decidable language. Let $a_n$ be the $n$th word in $A$ with respect to the lexicographic ordering. Then $K(a_n) \leq \log n + O(1)$.

Proof. Consider the following Turing machine:

---

**Input:** $\text{bin(n)}$

1. Enumerate $\{0,1\}^*$ in the lexicographic order. Let $x$ be the current word.
2. Since $A$ is decidable, we can check whether $x$ is in $A$.
3. Repeat this process until $n$ words in $A$ have been found.
4. Print $x$. 

© Markus Bläser 2007–2015
This Turing machine finds $a_n$ on input $\text{bin}(n)$. Let $e$ be the Gödel number of this machine. By construction, $U_{TM}$ on input $[e, \text{bin}(n)]$ outputs $a_n$. 

**Remark C.10** If there is an $\epsilon > 0$ such that $|A \cap \{0, 1\}^n| \leq (2 - \epsilon)^n$ for all $n$, then no long enough $x$ in $A$ can be Kolmogorov random.

### C.3 Lower bounds for 1-tape Turing machines

In the exercises, we constructed a 1-tape Turing machine that on input $x\#$ produces $x\#x$, i.e., it copies $x$ to the right of $. The number of steps of such a Turing machine is $\epsilon \cdot n^2$ for some $\epsilon > 0$.

We can use Kolmogorov complexity to prove that this is best possible. Let $M$ be a Turing machine that on input $x\#$ produces $x\#x$. Let $q$ be the number of states of $M$. Let $y$ be a string of length $n$. Consider the string $y0^n\#$. Let $t$ be the number of steps that $M$ computes on $y0^n\#$.

Consider the computation on $y0^n\#$. For each cell $i$ with $n + 1 \leq i \leq 2n$, we write down the sequence of states of $M$, when $M$ moves its head from cell $i$ to cell $i + 1$.

**Lemma C.11** If $q_1, \ldots, q_s$ is such a sequence of states as described above, then $K(y) \leq s \cdot \log q + \log \log q + \log n + O(1)$.

**Proof.** Assume that $q_1, \ldots, q_s$ is the sequence between the $(n + j)$th and $(n + j + 1)$th cell. The following Turing machine $N$ outputs $y$ on input $q_1, \ldots, q_s$ and $\log j$.

**Input:** $q_1, \ldots, q_s, \log j$

1. Print $0^{n-j+1}\#$ on the tape
2. For $i = 1, \ldots, s$ start on the leftmost 0 and simulate $M$ starting in state $q_i$ until $M$ enters the cell left to the leftmost 0.
3. Remove all unnecessary 0’s and #’s.

$N$ simulates the parts of the computation of $M$ to the right of the $(n+j)$th cell. When $M$ moves from the $(n+j)$th cell to the right, the only thing that determines its behaviour is the state. (Here we use that $M$ has only one tape!). Thus $N$ will produce $0^{n-j+1}\#y0^n$ in step 2. In step 3, $N$ just discards all the 0’s and the #. Thus it produces $y$ in the end.

The length of the input is $s \cdot \log q + \log \log q$ for the states (we can use a fixed length code with words of length $q$) plus $\log n$ for the cell number.
Lemma C.12 For each \( n + 1 \leq i \leq 2n \), let \( s_i \) be the length of the sequence of states between cell \( i \) and \( i + 1 \). There is an \( n + 1 \leq j \leq 2n \) such that \( s_j \leq t/n \), where \( t \) is the total number of steps of \( M \) on \( y0^n\# \).

Proof. Every state in a sequence of states corresponds to one step of the Turing machine. Hence \( t \geq \sum_{i=n+1}^{2n} s_i \).

Now take a Kolmogorov random \( y \), i.e., \( K(y) \geq n \). Let \( j \) as in Lemma C.12. By Lemmas C.11 and C.12,

\[
t \geq n \cdot s_j \geq \frac{K(y) - \log \log q - \log n - O(1)}{\log q} \geq \frac{n^2}{\log q} - \log n - O(1).
\]

Theorem C.13 For every 1-tape Turing machine \( M \) that solves the copy problem, there is an \( \epsilon > 0 \) such that \( M \) makes at least \( \epsilon \cdot n^2 \) steps in the worst case on inputs of length \( n \).

C.4 Undecidability of the Kolmogorov Complexity

The halting problem was our “starting” undecidable problem. All other results about non-computability followed by reduction. Kolmogorov complexity is another natural problem for which it is easy to give a direct proof that it is not Turing computable.

Theorem C.14 \( K \) is not Turing computable.

Proof. Assume on the contrary that there is a Turing machine \( M \) that computes \( K \). Let \( x_n \) be the first word with respect to the lexicographic order such that \( K(x_n) \geq n \). (\( x_n \) exists, since Kolmogorov random words have arbitrarily high complexity.)

We construct a Turing machine \( N \) that on input \( \text{bin} \ n \), outputs \( x_n \) as follows:

Input: \( \text{bin} \ n \)

1. Enumerate \( \{0, 1\}^* \) in lexicographic order. Let \( x \) be the current string.
2. Compute \( K(x) \) using \( M \).
3. If \( K(x) < n \), then go on with the next \( x \).
4. Otherwise, print \( x \).
By construction, $N$ on input $bin\ n$ produces the lexicographically first string whose Kolmogorov complexity is at least $n$. Let $e$ be an index of $N$. On input $[e,\ bin\ n]$, $U_{TM}$ prints $x_n$. Hence

$$K(x_n) \leq |[e,\ bin\ n]| \leq \log n + O(1).$$

This contradicts $K(x_n) \geq n$ for large enough $n$. ■

$K$ is a function and not a language but we easily get a language out of it.

**Exercise C.3** Show that the language $\{[x,\ cod(k)] \mid K(x) = k\}$ is not decidable.

$K$ is not harder then $H$, since we can show the following result.

**Theorem C.15** If $H \in \text{REC}$, then $K$ is Turing computable.

*Proof.* Let $M$ be a Turing machine that decides $H$. The following Turing machine $N$ computes $K(x)$:

---

**Input:** $x \in \{0,1\}^*$

1. Enumerate $\{0,1\}^*$ in lexicographic order. Let $y$ be the current string.
2. Use $M$ to decide whether $U_{TM}$ halts on $y$.
3. If yes, simulate $U_{TM}$ on $y$. Let $z$ be the output.
4. If $z \neq x$, then go on with the next $y$.
5. Return $|z|$.

---

$N$ always terminates since in step 2, we check whether $U_{TM}$ will halt and since there is always a string of length $|x| + O(1)$ such that $U_{TM}$ outputs $x$. By construction, $N$ finds a shortest string $y$ such that $U_{TM}$ on input $y$ outputs $x$. The length $|y|$ of this is $K(x)$. ■
Part II

Complexity
16 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.

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 “intractable” problems). Space consumption is another resource that we will investigate.

16.1 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 $SC(x) \vdash^*_M C_t$. Then, by definition, there is a unique sequence

$$SC(x) \vdash_M C_1 \vdash_M \cdots \vdash_M C_t.$$ 

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 $\text{Time}_M(x)$. If $M$ does not halt on $x$, then the computation of $M$ on $x$ is infinite. In this case $\text{Time}_M(x)$ is infinite.

For an $n \in \mathbb{N}$, we define the time complexity of $M$ as

$$\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid |x| = n\}.$$ 

In other words, $\text{Time}_M(n)$ measures the worst case 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 16.1** Consider the language 

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

$L$ can be recognized with space $O(\log n)$. 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 $\log n$ 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 $\text{Time}_M(n) \leq t(n)$ 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 16.2** Let $t : \mathbb{N} \to \mathbb{N}$. Then

$D\text{Time}(t) = \{ L \mid L \text{ is deterministically } t \text{ time decidable} \}$,

$D\text{Time}_k(t) = \{ L \mid \text{there is a } t \text{ time bounded } k\text{-tape Turing machine } M \text{ with } L = L(M) \}$.

© Markus Bläser 2007–2015
For a set of functions $T$, $\text{DTime}(T) = \bigcup_{t \in T} \text{DTime}(t)$. $\text{DTime}_k(T)$ 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 chapter 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. We will use a Turing machine $M$ with an extra input tape.

**Definition 16.3** Let $s : \mathbb{N} \to \mathbb{N}$. Then

$$\text{DSpace}(s) = \{L \mid L \text{ is deterministically } s \text{ space recognizable}\},$$

$$\text{DSpace}_k(s) = \{L \mid \text{there is a } s \text{ space bounded } k\text{-tape Turing machine } M \text{ with } L = L(M)\}.$$  

In the definition of $\text{DSpace}(s)$, the Turing machines have an additional input tape.

For a set of functions $S$, $\text{DSpace}(S) = \bigcup_{s \in S} \text{DSpace}(s)$. $\text{DSpace}_k(S)$ is defined analogously.

**Exercise 16.1** Intuitively, it is clear that sublinear time is not very meaningful here.$^1$ 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 $L(M) = \bigcup_{i=1}^m a_i \{0, 1\}^*$.  

16.2 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

$$\delta : Q \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^k \times \{L, S, R\}^k).$$

---

$^1$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.
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.

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 \( SC(x) \). 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 \( \ell \) 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 neither accepting nor 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

\[
L(M) = \{ x \in \Sigma^* \mid \text{ } \}
\]

**Example 16.4** Figure 16.1 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 16.2 shows the computation tree on input 010. It is quite easy to see that \( L(M) = \{ x \mid x \text{ contains at least one 0} \} \).

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 \( \infty \) otherwise. We set

\[
\text{Time}_M(n) = \max\{ \text{Time}_M(x) \mid |x| = n, \ x \in L(M) \}.
\]
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 16.2** 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 $O(t)$ 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” time bounds.

**Definition 16.5** Let $t : \mathbb{N} \to \mathbb{N}$. Then

\[
\text{NTime}(t) = \{ L \mid \text{there is a weakly } t \text{ time bounded nondeterministic Turing machine } M \text{ with } L = L(M) \},
\]

\[
\text{NTime}_k(t) = \{ L \mid \text{there is a weakly } t \text{ time bounded nondeterministic } k\text{-tape Turing machine } M \text{ with } L = L(M) \}.
\]

---

2One 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.
For a set of functions $T$, $\text{NTime}(T) = \bigcup_{t \in T} \text{NTime}(t)$. $\text{NTime}_k(T)$ is defined analogously.

**Warning!** I am fully aware of the fact that there does not exist a physical realization of a nondeterministic Turing machine! (At least, I do not know of any.) Nondeterministic Turing machines 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 16.3 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 $\infty$ 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 bounded** if $\text{Space}_M(n) \le s(n)$ for all $n$. We define **strongly $s$ space bounded** in the same way as we did for strongly time bounded.

**Definition 16.6** Let $s : \mathbb{N} \to \mathbb{N}$. Then

$$\text{NSpace}(s) = \{L \mid \text{there is a weakly } s \text{ space bounded nondeterministic Turing machine } M \text{ with } L = L(M)\},$$

$$\text{NSpace}_k(s) = \{L \mid \text{there is a weakly } s \text{ space bounded nondeterministic } k\text{-tape Turing machine } M \text{ with } L = L(M)\}.$$ 

*In the case of $\text{NSpace}(s)$, the Turing machines have an extra input tape.*

### 16.3 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, $O(\ell^3)$\(^\text{3}\) where $\ell$ 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 $X = \{x_1, x_2, \ldots\}$ 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.

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 occurrence 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 $\text{im } a \subseteq S$.

We can encode arithmetic formulas as follows: For instance, we can encode the variable $x$ by $0 \text{ bin}(i)$ and a constant $n$ by $1 \sigma(z) \text{ bin}(|z|)$ where $\sigma(z)$ is 1 if $z \geq 0$ and 0 otherwise. Then we define the encoding $c$ inductively by $c(F \cdot G) = [0, c(F), c(G)]$ and $c(F + G) = [1, c(F), c(G)]$. This is a very structured encoding, since it explicitly stores the order in which operations are performed. Alternatively, we first encode $x$ by the string $x \text{ bin}(i)$ and $z$ by $\sigma(z) \text{ bin}(|z|)$. 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 $O(\ell^3)$.

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

---

3$O(\ell^3)$ can be easily achieved for reasonable encodings: A formula $F$ of length $\ell$ has at most $\ell$ arithmetic operations and the value of the formula in the end has at most $\ell$ 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 $\ell$ of them. Using more sophisticated methods and a better analysis, one can bring down the evaluation time to $O(\ell^{1+\epsilon})$ for any $\epsilon > 0$. 

© Markus Bläser 2007–2015
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

$$AFSAT = \{ F \mid \text{there is an } \{0, 1\} \text{ assignment such that } F \text{ evaluates to 1.} \}$$

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 $F \in AFSAT$? 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 $O(n^3)$ whether $F$ evaluates to 1 or not. Thus

$$AFSAT \in \text{DTime}(O(2^n \cdot n^3))$$

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 16.3. 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.

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

$$AFSAT \in \text{NTime}(O(n^3))$$

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 $\text{NP}$-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 give nondeterminism its right to exists ... (and there might be some other reasons, too)

Excursus: Complexity of WHILE programs

© Markus Bläser 2007–2015
16. Turing machines and complexity classes

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 $|\text{cod}^{-1}(n)| = \log n$. 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 $\ell$ is the maximum over the running times on inputs $i$ with $|\text{cod}^{-1}(i)| = \ell$. If $S = (\sigma_0, \ldots, \sigma_\ell)$ is a state of a WHILE program, then the space used by this state is $\text{Space}(S) := \max\{|\text{cod}^{-1}(\sigma_\lambda)| \mid 0 \leq \lambda \leq \ell\}$.

Let $P$ be a simple statement. Assume we have a state $S = (\sigma_0, \ldots, \sigma_\ell)$ and let $S' = (\tau_0, \ldots, \tau_\ell) := \Phi_P(S)$. We claim that $\text{Space}(S') \leq 1 + \text{Space}(S)$, if $\text{Space}(S)$ 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 $|\text{cod}^{-1}(\tau'_i)| \leq 1 + \max\{|\text{cod}^{-1}(\sigma_j)|, |\text{cod}^{-1}(\sigma_k)|\}$. The same is true if $P$ is a subtraction. The case $P = x_i := c$ can only change $\text{Space}(S)$ is $\text{Space}(S) \leq |\text{cod}^{-1}(c)|$. 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 program 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 $<\ldots>$, 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,
16.3. An example

we again only get a quadratic slowdown.
17 Tape reduction, compression, and acceleration

In this chapter, we further investigate the Turing machine model before we come to results that also hold in other models of computation in the next chapters.

17.1 Tape reduction

Definition 17.1 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 17.2 Every deterministic Turing machine can be simulated by a deterministic 1-tape Turing machine.

Remark 17.3 The construction is quite similar to the universal Turing machine that we constructed in the first part of this lecture.

Proof. Let $M$ be a $k$-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 $(2k - 1)$th component of a symbol of $\Gamma'$ stores the content of the $k$th tape of $M$. The $2k$th component is used to mark the position of the head on the $k$th tape. There will be exactly one $*$ on the $2k$ track, this $*$ will mark the position of the head. All other entries of the track will be filled with $-$'s. Figure 17.1 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 17.2 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 leftmost position of the tape visited so far and moves to the right until it reaches...
17.1. Tape reduction

Figure 17.1: 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 "*".

the first blank (of $S$). 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, 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 17.4 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 17.5 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 17.6 (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.

1A blank of $S$ indicates that we reached a position that $M$ has not visited so far on any of its tapes.
Input: $x \in \Sigma^*$
Output: accept if $x \in L(M)$, reject otherwise

1. $S$ replaces the input $x$ by the corresponding symbol of $\Gamma'$, i.e., $x_1$ is replaced by $(x_1, *, \Box, *, \ldots, \Box, *)$ and each other $x_\nu$ is replaced by $(x_\nu, -, \Box, -, \ldots, \Box, -)$.

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 on of the markers * to a cell that still contains $\Box$, the blank of $S$, then it first replaces this blank by $(\Box, -, \ldots, \Box, -)$.

4. If $M$ accepts, $S$ accepts. Otherwise, $S$ rejects.

Figure 17.2: The simulator $S$. 
But we tacitly 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 \{collect\} × $Q$ × $(\Gamma \cup \{/\})^k$. 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 this first component.) The second component stores the current state of $M$. It shall not be changed during the collection phase. Finally, $\gamma_\kappa$ stores the symbol that is read by $M$ on the $\kappa$th tape. $\gamma_\kappa /=\$ indicates that the position of the head on tape $\kappa$ has not been found yet.

The transition function $\delta'$ 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 $2\kappa$th track is $\$, then we found the head on the $\kappa$th tape and store $\eta_{2\kappa-1}$, the symbol that $M$ reads on the $\kappa$th tape, in the state of $S$.

**Definition 17.7** Let $t, s : \mathbb{N} \to \mathbb{N}$. Then

$$\text{DTimeSpace}(t, s) = \{ L \mid \text{there is a } t \text{ time and } s \text{ space bounded Turing machine } M \text{ with } L = L(M) \}.$$

\(\text{DTimeSpace}_k(t, s)\) and \(\text{NTimeSpace}_k(t, s)\) are defined accordingly.

**Theorem 17.8** For all $t, s : \mathbb{N} \to \mathbb{N}$,

$$\text{DTimeSpace}(t, s) \subseteq \text{DTimeSpace}_1(O(ts), O(s)),$$

$$\text{NTimeSpace}(t, s) \subseteq \text{NTimeSpace}_1(O(ts), O(s)).$$

**Proof.** Let $L \in \text{DTimeSpace}(t, s)$. Let $M$ be a deterministic Turing machine with $L = L(M)$. Assume that $M$ has $k$ tapes. The Turing machine $S$ in Theorem 17.2 simulates one step of $M$ by $O(s(|x|))$ 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 17.4. □
Corollary 17.9  For all $t : \mathbb{N} \to \mathbb{N}$,
\[
\text{DTime}(t) \subseteq \text{DTime}_1(O(t^2)),
\]
\[
\text{NTime}(t) \subseteq \text{NTime}_1(O(t^2)).
\]

Proof. Let $M$ be a $t$ time bounded Turing machine. In $t$ steps, a turing machine can visit at most $t$ cells. Thus $\text{Space}_M(n) \leq t(n)$ for all $n$, and the corollary follows from Theorem 17.8. $\blacksquare$

17.2 Tape compression

The aim of this and the next section is to show that we do not need to take care of constant factors. 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 17.10  For all $0 < \epsilon \leq 1$ and all $s : \mathbb{N} \to \mathbb{N}$,
\[
\text{DSpace}(s(n)) \subseteq \text{DSpace}_{1,E}(\lceil \epsilon s(n) \rceil)
\]
\[
\text{NSpace}(s(n)) \subseteq \text{NSpace}_{1,E}(\lceil \epsilon s(n) \rceil)
\]

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 $\Gamma$. 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. $\blacksquare$

Remark 17.11 (Implementation details) Again, let's try to formalize a part of the transition function $\delta'$ of $S$. The states of $S$ are of the form

© Markus Bläser 2007–2015
\[ Q \times \{1, \ldots, c\}. \] \((q, i)\) means that \(M\) is in state \(q\) and its head is on the \(i\)th 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 17.1** Show the following “converse” of Theorem 17.10: For any \(s\) space and \(t\) time bounded Turing machine \(M\) with input alphabet \(\{0, 1\}\), there is a \(O(s)\) space and \(O(t)\) time bounded Turing machine that only uses the work alphabet \(\{0, 1, \square\}\).

## 17.3 Acceleration

Next, we prove a similar speed up for time. This simulation is a little more complicated than the previous one.

**Exercise 17.2** Show the following: For all \(k \geq 2\), all \(t : \mathbb{N} \to \mathbb{N}\), and all \(0 < \epsilon \leq 1\),

\[
DTime_k(t(n)) \subseteq DTime_k(n + \epsilon(n + t(n)))
\]

\[
NTime_k(t(n)) \subseteq NTime_k(n + \epsilon(n + t(n))).
\]

**Proof overview:** Like in Theorem 17.10, 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 \(\epsilon\) in Exercise 17.2. If \(t(n) = O(n)\), 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!
17.4 Further exercises

Exercise 17.3 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

$$
\begin{align*}
\text{Time}_S(n) &= \begin{cases} 
n & \text{if } n \leq c \\
\text{Time}_M(n) + c & \text{otherwise}
\end{cases} \\
\text{Space}_S(n) &= \begin{cases} 
0 & \text{if } n \leq c \\
\text{Space}_M(n) & \text{otherwise}
\end{cases}
\end{align*}
$$

In other words, only the asymptotic behaviour matters.
18 Space versus Time, Nondeterminism versus Determinism

18.1 Constructible functions

Definition 18.1 Let \( s, t : \mathbb{N} \rightarrow \mathbb{N} \).

1. \( t \) is time constructible if there is a \( O(t) \) time bounded deterministic Turing machine \( M \) that computes the function \( 1^n \mapsto \text{bin}(t(n)) \).

2. \( s \) is space constructible if there is an \( O(s) \) space bounded deterministic Turing machine \( M \) (with extra input tape) that computes the function \( 1^n \mapsto \text{bin}(s(n)) \).

Above, \( \text{bin}(n) \) denotes the binary representation of \( n \).

Exercise 18.1 Show the following:

1. If \( t \) is time constructible, then there is a \( O(t) \) time bounded deterministic Turing machine that on input \( x \) writes \( 1^{|x|} \) 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 \( 1^{|x|} \) on one of its tapes.

Time and space constructible functions “behave well”. One examples for this is the following result.

Lemma 18.2 Let \( t \) be time constructible and \( s \) be space constructible.

1. If \( L \in \text{NTime}(t) \) then there is a strongly \( O(t) \) time bounded nondeterministic Turing machine \( N \) with \( L = L(N) \).

2. If \( L \in \text{NSpace}(s) \), then there is a strongly \( O(s) \) space bounded nondeterministic 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 $O(t)$ time bounded, since counting to $t(|x|)$ in binary can be
 done in $O(t(|x|))$ 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 18.1), $s(|x|)$ 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 $O(s)$ 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 18.2** 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 \( f \in \Omega(\log n) \).

### 18.2 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 \( \text{Conf}_M \). We denote this graph by \( CG_M = (\text{Conf}_M, \vdash_M) \).

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 \( SC_M(x) \). This task is undecidable in general 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 18.3

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 \( SC(x) \).

**Proof.** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}}) \) 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

\[
|Q| \cdot (|\Gamma|^{s(|x|)})^k \cdot s(|x|)^k \cdot (|x| + 2)
\]  

(18.1)

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 (18.1) 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 18.3

Give an upper bound for the constant \( c \) above.

#### Corollary 18.4

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 18.3.

**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. 

---

\(^1\)Note that the binary relation \( \vdash_M \) is nothing else than a set of pairs of elements from \( \text{Conf}_M \), that is, directed edges.
Corollary 18.5 Let \( s(n) \geq \log n \) be space constructible. Then \( \text{DSpace}(s) \) is closed under complement, i.e., if \( L \in \text{DSpace}(s) \) so is \( \overline{L} \).

Proof. Let \( M \) be an \( s \) space bounded deterministic Turing machine for \( L \). We construct a deterministic Turing machine \( \overline{M} \) as follows: \( \overline{M} \) simulates \( M \) step by step. If \( M \) halts, then \( \overline{M} \) halts, too, and accepts if \( M \) rejects and vice versa. Problematic is the case when \( M \) does not halt. Here \( \overline{M} \) 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 \( \overline{M} \) stops and accepts.

Remark 18.6 Corollary 18.5 is trivially true for deterministic time classes, we just have to exchange accepting states with rejecting states.

Exercise 18.4 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 18.7 Let \( G \) be a graph. If there is a path of length \( \ell \) 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 18.8 Let \( M \) be an \( s \) space bounded (deterministic or nondeterministic) Turing machine where \( s(n) \geq \log n \) is space constructible. There is a \( 2^{O(s(|x|))} \) time bounded deterministic Turing machine \( M_1 \) and \( O(s^2(|x|)) \) space bounded deterministic Turing machine \( M_2 \) that given \( x \) decides whether an accepting configuration in \( \text{CG}_M \) is reachable from \( \text{SC}_M(x) \).

Proof. To achieve the time bound \( 2^{O(s(|x|))} \), \( M_1 \) 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 \( \text{CG}_M \), 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 18.3, it is now sufficient to check whether \( M_1 \) could

\(^2\)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.
reach an accepting configuration from $\text{SC}(x)$. 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, 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 time $2^{O(s(|x|))}$ and space $O(s^2(|x|))$, then we are done. Here $c$ is
the constant from Lemma 18.3. We enumerate all accepting configurations $C$—there are at most $c^{s(|n|)}$—and compute $R(\text{SC}(x), C, c^{s(n)})$. We accept iff at least one of these values is one. Note that we can reuse the space when computing $R(\text{SC}(x), C, c^{s(n)})$ for the next $C$.

We will compute $R(C, C', \ell)$ recursively. We use the identity

$$R(C, C', \ell) = 1 \iff \text{there is a configuration } C'' \text{ such that } R(C, C'', \lceil \ell/2 \rceil) = R(C'', C', \lceil \ell/2 \rceil) = 1 \text{ or } C' \vdash M C'.$$

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 $R(C'', C', \lceil \ell/2 \rceil)$. 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

$$S(\ell) \leq O(s(|x|)) + S(\lceil \ell/2 \rceil),$$

$$S(1) \leq O(s(|x|)).$$

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, $S(c^{s(|x|)}) = O(s^2(|x|))$. \(\blacksquare\)

## 18.3 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 18.9** Let $s(n) \geq \log(n)$ be space constructible. Then

$$\text{DSPACE}(s) \subseteq \text{NSPACE}(s) \subseteq \text{DTIME}(2^{O(s(n))}).$$

**Proof.** The first inclusion is trivial. The second inclusion follows from
Lemma 18.8. \(\blacksquare\)
18.4 Nondeterminism versus determinism

Theorem 18.10 Let $t$ be time constructible. Then
\[ \text{NTime}(t) \subseteq \text{NSpace}(t) \subseteq \text{DTime}(2^{O(t)}). \]

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 18.8. ■

Theorem 18.11 (Savitch) Let $s(n) \geq \log n$ be space constructible. Then
\[ \text{NSpace}(s) \subseteq \text{DSpace}(O(s^2)). \]

Proof. Again, this follows from Lemma 18.8. ■
19  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 17.8).

19.1  A technical lemma

Lemma 19.1  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 \( i \)th symbol of the work alphabet
  is represented by the string \( \text{bin}(i) \) 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 $g \in \text{im göd}_{TM}$.

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 $O(t_2)$ 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. \(\blacksquare\)

### 19.2 Deterministic hierarchy theorems

**Theorem 19.2 (Deterministic space hierarchy)** Let $s_2(n) \geq \log n$ be space constructible and $s_1(n) = o(s_2(n))$. Then
\[
\text{DSpace}(s_1) \subsetneq \text{DSpace}(s_2).
\]

**Proof.** Consider $C_1$ from Lemma 19.1. $L(C_1) \in \text{DSpace}(s_2)$. 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 17.8. Thus there is also no $s_1$ space bounded deterministic k-tape Turing machine M such that $L(M) = L(C_1)$.

\(\blacksquare\)

\(^1\)We cannot bound $|x| + 2$ by $3^{s_1(|x|)}$, 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 19.3 (Deterministic time hierarchy)** Let $t_2$ be time constructible and $t_1^2 = o(t_2)$. Then

$$\text{DTime}(t_1) \subsetneq \text{DTime}(t_2).$$

**Proof.** Consider $C_2$ from Lemma 19.1. $L(C_2) \in \text{DTime}(t_2)$. 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 17.8. Thus there is also no $t_1$ time bounded deterministic $k$-tape Turing machine $M$ such that $L(M) = L(C_1)$.

**19.3 Remarks**

The assumption $t_1^2 = o(t_2)$ in the proof of the time hierarchy theorem is needed, since we incur a quadratic slowdown when simulating $k$-tape Turing machines by 1-tape Turing machines.

Hennie and Stearns showed the following theorem.

**Theorem 19.4 (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 19.5** Let $t_2$ be time constructible and $t_1 \log t_1 = o(t_2)$. Then

$$\text{DTime}(t_1) \subsetneq \text{DTime}(t_2).$$

The answer to the following question is not known.$^2$

**Research Problem 19.1** Can the assumption $t_1 \log t_1 = o(t_2)$ be further weakened? In particular, can we get a better simulation of arbitrary deterministic 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 19.6 (Fürer) Let \( k \geq 2 \), \( t_2 \) time constructible, and \( t_1 = o(t_2) \). Then

\[
\text{DTIME}_k(t_1) \subsetneq \text{DTIME}_k(t_2).
\]

We conclude with pointing out that the assumption that \( s_2 \) and \( t_2 \) are constructible are really necessary.

Theorem 19.7 (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

\[
\text{DTIME}(f(t(n))) = \text{DTIME}(t(n)),
\]

\[
\text{DSPACE}(f(s(n))) = \text{DSPACE}(s(n)).
\]

Proof. We only construct \( t \), the construction of \( s \) is similar. Let \( T_g(n) := \text{Time}_{\text{göd}^{-1}(g)}(n) \) 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 \( \text{cod}(g) \leq n \),

\[
T_g(n) \leq f(m) \Rightarrow T_g(n) \leq m.
\]

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 \( \text{cod}(g) \leq n \). 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 \( \text{göd}^{-1}(g) \) 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 \( g = \text{göd}_T(M) \). By the construction of \( t \), \( T_g(n) \leq t(n) \) for all \( n \geq \text{cod}(g) \). Therefore, \( L(M) \in \text{DTIME}(t) \). Thus, \( \text{DTIME}(f \circ t) = \text{DTIME}(t) \).

Set for instance \( g(n) = 2^n \) (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:

\[
\text{NSPACE}(s_1) \subseteq \text{DSPACE}(s^2_1) \subseteq \text{DSPACE}(s^2_2) \subseteq \text{NSPACE}(s^2_2)
\]

for any functions \( s_1, s_2 : \mathbb{N} \to \mathbb{N} \) with \( s_1 = o(s_2) \), \( s_1 \) and \( s_2 \) space constructible, and \( s_1(n) \geq \log n \). It is even possible to show a tight hierarchy like in Theorem 19.2. This uses the non-trivial—and unexpected—fact that \( \text{NSPACE}(s) \) 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

$$\text{NTime}(t_2) \setminus \text{NTime}(t_1) \neq \emptyset.$$ 

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.
20  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 20.1**

\[
P = \bigcup_{i \in \mathbb{N}} \text{DTime}(O(n^i))
\]

\[
\text{NP} = \bigcup_{i \in \mathbb{N}} \text{NTime}(O(n^i))
\]

\(P\) (\(P\) stands for *polynomial time*) is the class of problems that are considered to be feasible or tractable. Frankly, an algorithm with running time \(O(n^{1024})\) 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.

\(\text{NP}\) (\(\text{NP}\) 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 \(\text{NP}\) 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 16.) This is also true for \(\text{NP}\), if we equip WHILE programs with nondeterminism in a suitable way.

The question whether \(P = \text{NP}\) 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

\[
\text{NP} = \bigcup_{i \in \mathbb{N}} \text{NTime}(O(n^i)) \subseteq \bigcup_{i \in \mathbb{N}} \text{DTime}(2^{O(n^i)}) =: \text{EXP},
\]

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.

20.1 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 $t$:

$$s$-$t$-CONN = \{(G, s, t) \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

$(G, s, t)$ 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 20.2 $s$-$t$-CONN $\in P$.

20.2 $NP$ 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 20.3 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 $x \not\in L$ 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 20.4** $L \in \text{NP}$ iff there is a polynomial time verifier for $L$.

**Proof.** We only prove the “$\Rightarrow$”-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 $\delta$ 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 $x \not\in L$, 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 20.1** Prove the other direction of Theorem 20.4

### 20.3 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 $\pi$ such that for all $1 \leq i < n$, $\{ v_{\pi(i)}, v_{\pi(i+1)} \} \in E$ and $\{ v_{n(1)}, v_{\pi(1)} \} \in E$, 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., $\sum_{i=1}^{n-1} w(\{ v_{\pi(i)}, v_{\pi(i+1)} \}) + w(\{ v_{n(1)}, v_{\pi(1)} \})$. 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 $\overline{x}_i$. A **clause** is a disjunction of literals $\ell_1 \lor \cdots \lor \ell_k$. $k$ is the length of the clause. A **formula in conjunctive normal form** (formula in CNF for short) is a conjunction of clauses $c_1 \land \cdots \land c_m$. 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 $\textbf{NP}$, is the following problem:

$$\text{SAT} = \{ \phi \mid \phi \text{ is a satisfiable formula in CNF} \}.$$
A formula in CNF is in \(\ell\)-CNF, if all its clauses have length at most \(\ell\).

\(\ell\)SAT is the following problem:

\[
\ell\text{SAT} = \{ \phi \mid \phi \text{ is a satisfiable formula in } \ell\text{-CNF} \}.
\]

**Golden rule of nondeterminism**

*Non-determinism is interesting because it characterizes important problems.*

We do not know any physical equivalent to nondeterminism. As far as I know, nobody has built a nondeterministic Turing machine. But \(\text{NP}\) is an interesting class because it contains a lot of important problems.

**Theorem 20.5** Clique, VC, Subset-Sum, HC, TSP, SAT, \(\ell\)SAT \(\in\) \(\text{NP}\).

**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 \(\ell\)SAT, 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. \(\blacksquare\)
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.
21 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 \( L \in NP \) such that \( L \notin P \). 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.

21.1 Polynomial time reductions

Definition 21.1 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 \( L \leq_P L' \).

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 21.2 If \( L \leq_P L' \) and \( L' \in P \), then \( L \in P \).

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 21.3 \( \leq_P \) 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 21.2. \( \blacksquare \)

**Polynomial time many one reductions versus recursive many one reductions**

A many one reduction \( f \) from \( L \) to \( L' \) has the following property:
\[
x \in L \iff f(x) \in L' \text{ for all } x \in \{0, 1\}^*.
\]

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 \( L' \in \text{REC} \) (or \( \text{RE} \)), then \( L \in \text{REC} \) (or \( \text{RE} \)).

Important properties of polynomial time many one reductions:

- \( \leq_P \) is transitive.
- If \( L \leq_P L' \) and \( L' \in \text{P} \), then \( L \in \text{P} \).

### 21.2 NP-complete problems

**Definition 21.4**  
1. A language \( L \) is **NP-hard** if for all \( L' \in \text{NP} \), \( L' \leq_P L \).
2. \( L \) is called **NP-complete**, if \( L \) is NP-hard and \( L \in \text{NP} \).

**Lemma 21.5** If \( L \) is NP-hard and \( L \in \text{P} \), then \( \text{P} = \text{NP} \).
Proof. Let $L' \in \text{NP}$. Since $L$ is NP-hard, $L' \leq_P L$. By Lemma 21.2, $L' \in \text{P}$. ■

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 21.6 If $L$ is NP-hard and $L \leq_P L'$, then $L'$ is NP-hard.

Proof. Let $L'' \in \text{NP}$. Since $L$ is NP-hard, $L'' \leq_P L$. Since $\leq_P$ is transitive, $L'' \leq_P L'$. 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 21.7 (Cook–Karp–Levin) 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 courser 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 21.8 SAT $\leq$ AFSAT.

Proof. Let $\phi$ be a Boolean formula in CNF with $n$ variables. We construct an arithmetic formula $F_\phi$ such that every satisfying assignment $a \in \{0, 1\}^n$ of $\phi$ 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 $\ell$ be a literal. If $\ell = x$, then $F_\ell = x$. If $\ell = \bar{x}$, then $F_\ell = 1 - x$.

If $c = \ell_1 \lor \cdots \lor \ell_k$ 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 $F_{\ell_i}$ evaluates to 1.

1This 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.

© Markus Bläser 2007–2015
Finally, if \( \phi = c_1 \land \cdots \land c_m \) is a conjunction of clauses, then \( F_\phi = F_{c_1} \cdots F_{c_m} \). \( F_\phi \) 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.

Let’s start to show that the other problems introduced in this chapter are all \( \text{NP} \)-complete.

**Lemma 21.9** For all \( \ell \geq 3 \), \( \ell\text{SAT} \) is \( \text{NP} \)-complete. \(^2\)

**Proof.** We show that \( \text{SAT} \leq \ell\text{SAT} \). It suffices to show this for \( \ell = 3 \). Let \( \phi \) be a formula in CNF. We have to map \( \phi \) to a formula \( \psi \) in 3-CNF such that \( \phi \) is satisfiable iff \( \psi \) is satisfiable.

We replace each clause \( c \) of length > 3 of \( \phi \) by a bunch of new clauses. Let \( c = \ell_1 \lor \cdots \lor \ell_k \) with literals \( \ell_k \). Let \( y_1, \ldots, y_{k-3} \) be new variables. (We need new variables for each clause.) We replace \( c \) by

\[
(\ell_1 \lor \ell_2 \lor y_1) \land (\bar{y}_1 \lor \ell_3 \lor y_2) \land \cdots \land (\bar{y}_{k-4} \lor \ell_{k-2} \lor y_{k-3}) \land (\bar{y}_{k-3} \lor \ell_{k-1} \lor \ell_k) \quad (21.1)
\]

If we do this for every clause of \( \phi \), we get the formula \( \psi \). This transformation is obviously polynomial time computable.

It remains to show that \( \phi \) is satisfiable iff \( \psi \) is satisfiable. Let \( a \) be some satisfying assignment for \( \phi \). We extend this assignment to a satisfying assignment of \( \psi \). Since \( a \) satisfies \( \phi \), for a given clause \( c \), there is one literal, say \( \ell_i \) 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 (21.1) are satisfied. Thus we found a satisfying assignment for \( \psi \). On the other hand, if \( \psi \) is satisfiable, then any satisfying assignment \( b \) that satisfies all the clauses in (21.1) has to set one \( \ell_i \) to 1, since the \( y_j \)’s can only satisfy at most \( k - 3 \) clauses but (21.1) contains \( k - 2 \) clauses. Thus the restriction of \( b \) to the variables of \( \phi \) satisfies \( c \) and henceforth \( \phi \).

**Exercise 21.1** Astonishingly, \( \text{2SAT} \in \text{P} \).

1. **Given a formula \( \phi \) 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 \( \ell_1 \lor \ell_2 \) we add the two edges \( (\bar{\ell}_1, \ell_2) \) and \( (\bar{\ell}_2, \ell_1) \) to \( E \). Show the following: \( \phi \) is satisfiable iff for all \( 1 \leq \nu \leq n \), there is no directed cycle that contains \( x_\nu \) and \( \bar{x}_\nu \).**

2. **Conclude that \( \text{2SAT} \in \text{P} \).**

\(^2\)Our proof of the Cook-Karp-Levin theorem will actually show that \( \text{3SAT} \) is \( \text{NP} \)-complete. It is nevertheless very instructive to see the reduction from \( \text{SAT} \) to \( \text{3SAT} \).
Reducing SAT to $\ell$-SAT was not too hard. (Or at least, it does not look too unreasonable that one can find such a reduction.) Reducing SAT or $\ell$-SAT 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 21.10** $\ell$-SAT $\leq_P$ Clique.

*Proof.* Let $\phi$ be a formula in 3-CNF. We may assume that each clause of $\phi$ has exactly three literals by possibly repeating some literals. Let $\phi = (\ell_{1,1} \lor \ell_{1,2} \lor \ell_{1,3}) \land \cdots \land (\ell_{m,1} \lor \ell_{m,2} \lor \ell_{m,3})$.

We have to construct a pair $(G,k)$ such that $G = (V,E)$ has a $k$-clique iff $\phi$ 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 $\ell_{i,s} \neq \ell_{j,t}$. 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 $\phi$ is satisfiable, then there is a satisfying assignment for $\phi$, 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 $\phi$ is satisfiable.

The mapping $\phi \mapsto (G,k)$, is obviously polynomial time computable. □

**Lemma 21.11** Clique $\leq_V$ VC.

*Proof.* For a graph $G = (V,E)$, let $\bar{G} = (V, \left\{(V \setminus E)\right\})$ 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 $\bar{G}$, 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 $\bar{G}$. □
Lemma 21.12 \( 3\text{SAT} \leq_P \text{Subset-Sum} \).

The proof of this lemma is more complicated; we defer it to the next chapter.

Exercise 21.2 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 \( P(\nu, \beta) = P(\nu - 1, \beta) \lor P(\nu - 1, \beta - x_{\nu}) \).
2. Design an algorithm with running time \( O(nb) \) for Subset-Sum.
3. Does this show that \( P = \text{NP} \)? (See also Chapter ??.)

Lemma 21.13 \( 3\text{SAT} \leq_P \text{HC} \).

This proof is again deferred to the next section.

Lemma 21.14 \( \text{HC} \leq_P \text{TSP} \).

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 \( n \). \( \blacksquare \)

Theorem 21.15 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 21.1: The reduction scheme. An arrow from $A$ to $B$ means that we proved $A \leq_P B$. 
22 More reductions

Only the result on Subset-Sum was discussed in class. The results on HC are not relevant for the exam.

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.

22.1 Subset-Sum

We first start with the proof of Lemma 21.12. As Exercise 21.2 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 21.12. 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 $\ell$ 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 $x_\nu$, let $c_{\mu_1}, \ldots, c_{\mu_s}$ be the clauses in which it appears positively, or in other words, $c_{\mu_1}, \ldots, c_{\mu_s}$ are the clauses that contain the literal $x_\nu$. Then

$$a(x_\nu) = 10^{\nu-1} + 10^n(10^{\mu_1-1} + \cdots + 10^{\mu_s-1}).$$

For a literal $\bar{x}_\nu$, let $c_{\bar{\mu}_1}, \ldots, c_{\bar{\mu}_s}$ are the clauses that contain the literal $\bar{x}_\nu$. Then

$$a(\bar{x}_\nu) = 10^{\nu-1} + 10^n(10^{\bar{\mu}_1-1} + \cdots + 10^{\bar{\mu}_s-1}).$$

Choosing $a(x_\nu)$ indicates that we set $x_\nu$ to 1. Choosing $a(\bar{x}_\nu)$ indicates that we set $\bar{x}_\nu$ to 1, i.e., $x_\nu$ to 0. Of course we can set $x_\nu$ 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 \(a(x_\nu),a(\overline{x}_\nu)\), \(1 \leq \nu \leq n\), 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 \(a(\overline{x}_\nu)\), \(1 \leq \nu \leq n\). 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 \(c_\mu\) 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” \(c_\mu,1, c_\mu,2\), \(1 \leq \mu \leq m\). 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 \(\phi\) has a satisfying assignment iff a subset of \(a(x_\nu),a(\overline{x}_\nu), 1 \leq \nu \leq n\), 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.

22.2 Hamiltonian Cycle

In order to prove Lemma 21.13, 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 visits each node exactly once and all the edges in the cycle have to point in the same direction.

\(\text{Dir-HC}\) is a generalization of \(\text{HC}\). 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 \(\text{HC} \leq_p \text{Dir-HC}\). But we can also show the converse.

Lemma 22.1 \(\text{Dir-HC} \leq_p \text{HC}\).

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...
Chapter 22. More reductions

Figure 22.1: 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, \( v, v_{\text{in}}, v_{\text{out}} \). For every directed edge \((x, y)\), we add the undirected edge \( \{x_{\text{out}}, y_{\text{in}}\} \).

Figure 22.2: The gadget for the reduction of 3SAT to Dir-HC.

three nodes \( v_{\text{in}}, v, v_{\text{out}} \) and connect \( v_{\text{in}} \) with \( v \) and \( v_{\text{out}} \) with \( v \). Then for every directed edge \((x, y) \in E\), we add the undirected edge \( \{x_{\text{out}}, y_{\text{in}}\} \) to \( E' \). Figure 22.1 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 \( G' \) 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.\footnote{Such a thing is usually called a \emph{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.}

© Markus Bläser 2007–2015
22.2. Hamiltonian Cycle

![Figure 22.3: The paths connecting the nodes in S.](image1)

![Figure 22.4: 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.](image2)

**Lemma 22.2** Let $G$ be the graph in Figure 22.2.

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.

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 22.3 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 22.4 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 22.1** Draw the corresponding figures for the pair $(a_2, b_1)$.

**Lemma 22.3** $3\text{SAT} \leq_P \text{Dir-HC}$.

**Proof.** For this proof, we have to do the following: Given a formula $\phi$ in 3-CNF, we have to map it to a directed graph $G$ (depending on $\phi$) such that $\phi$ is satisfiable iff $G$ has a Hamiltonian cycle. Every variable of $\phi$ will be represented by a node, every clause will be represented by a gadget from Figure 22.2.

Let $x_1, \ldots, x_n$ be the variables of $\phi$ 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 22.5 shows an example. \( x_2 \) is the first literal of \( c_3 \), the third of \( c_5 \), and the first of \( c_8 \). \( \overline{x}_1 \) is the second literal of \( c_2 \).

Let \( G \) be the graph constructed from \( \phi \). \( G \) can certainly constructed in polynomial time. So it remains to show that \( \phi \) has a satisfying assignment if \( G \) has a Hamiltonian cycle.

For the "\( \Rightarrow \)"-direction, let \( a \) be a satisfying assignment of \( \phi \). 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 22.2, 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 22.2, 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 occurrences of \( x_i \) or through the path with negative occurrences 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 22.4** \( \text{3SAT} \leq_P \text{HC} \).
Figure 22.5: 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 gadgets 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.
D 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 \( 3\text{SAT} \leq \text{p} \) Subset-Sum look like a picnic.) The general reduction scheme looks as follows:

\[
\begin{align*}
\text{Turing machines} \downarrow \text{oblivious Turing machines} \downarrow \\
\text{Boolean circuits} \downarrow \\
\text{CSAT} \downarrow \\
\text{SAT}
\end{align*}
\]

As an intermediate concept, we introduce \textit{Boolean circuits}. We show that Boolean circuits can simulate Turing machines. To do so, we have to first make the Turing machine \textit{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 \textit{circuit satisfiability problem} CSAT (given a circuit \( C \), does it have a satisfying assignment?) is NP-hard. Lastly, we show that CSAT \( \leq \text{p} \) SAT.

D.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 \textit{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 \(\lor\) or \(\land\) 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 \(\{0, 1\}^n \rightarrow \{0, 1\}^m\) 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 \(\text{val}(v, \xi) = 0\) or 1, respectively. This defines the values for all nodes of depth 0. Assume that the value of all nodes of depth 0 is 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 \(\lor\) or \(\land\) and \(u_1, u_2\) are the predecessors of \(v\), then \(\text{val}(v, \xi) = \text{val}(u_1, \xi) \lor \text{val}(u_2, \xi)\) or \(\text{val}(v, \xi) = \text{val}(u_1, \xi) \land \text{val}(u_2, \xi)\). For each node \(v\), this defines a function \(\{0, 1\}^n \rightarrow \{0, 1\}\) 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 \(\{0, 1\}^n \rightarrow \{0, 1\}^m\) 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 \(\{\neg, \lor, \land\}\). This set is also called standard basis. This standard is known to be complete, that is, for any Boolean function \(f : \{0, 1\}^n \rightarrow \{0, 1\}^m\), there is a 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 D.1** Every Boolean function \(f : \{0, 1\}^n \rightarrow \{0, 1\}\) can be computed by a Boolean circuit of size \(2^{O(n)}\). (Remark: This can be sharpened to \((1 + \varepsilon) \cdot 2^n/n\) for any \(\varepsilon > 0\). The latter bound is tight: For any \(\varepsilon > 0\) and any large enough \(n\), there is a Boolean function \(f : \{0, 1\}^n \rightarrow \{0, 1\}\) such that every circuit computing \(f\) has size \((1 - \varepsilon)2^n/n\).)
D.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., \( \{0,1\}^* \rightarrow \{0,1\}^* \). Boolean circuits only compute a function of fixed size \( \{0,1\}^n \rightarrow \{0,1\}^m \). 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 D.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 \( \{0,1\}^* \rightarrow \{0,1\} \) 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 family of Boolean circuits. To exclude such phenomena, we put some restrictions on the families.

**Definition D.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 \( C \) is constructible in time polynomial in \( n \).

Polynomial time uniform families of circuits always have polynomially bounded size.

D.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 D.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 D.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 17.2. 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 17.2.

To simulate one step of M, it makes a sweep over all the marked $t(|x|)$ cells and not just those visited 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 D.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...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...0 and 1 will be mapped to 1...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) $(q, \gamma)$ in to D, we get the output $(q', \gamma')$ 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 $i$th 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 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 D.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.

\[\text{D.4 The proof}\]

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 \( \xi \) with \( C(\xi) = 1 \).

**Theorem D.6** CSAT is NP-hard

**Proof.** Let \( L \in \text{NP} \) 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 D.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 D.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 $\Gamma$ 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 $x_\nu$ 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 $\text{CSAT}$. It is also polynomial time, since $C$ can be constructed in polynomial time.

**Theorem D.7** SAT is NP-hard.

**Proof.** It is sufficient to show $\text{CSAT} \leq \text{SAT}$. 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 exponential blow up.

Instead, we introduce new input variables $a_1, \ldots, a_s$. The idea is that $a_j$ is the output of the $j$th 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

$$(a_i \lor a_j) \land (\neg a_i \lor \neg a_j).$$

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

$$(\neg a_j \lor a_i) \land (\neg a_j \lor a_h) \land (a_j \lor \neg a_i \lor \neg a_h). \quad (D.1)$$

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 assignment to the input variables of $C$ such that $C$ evaluates to one. ■

**Remark D.8** The formula constructed in the proof above is actually in 3-CNFS. So we directly get that 3SAT is NP-complete.

**Exercise D.2** Construct a similar expression as in (D.1) for the Boolean or.
Part III

Formal languages
# 23 Finite automata and regular languages

## 23.1 The Chomsky hierarchy

In the 1950s, Noam Chomsky 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.

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 hierarchy contains four classes of languages that can be generated by four different kinds of grammars.

---

## 23.2 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.\(^1\) 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 inserted 40 cents so far, he can press the “Money back” button. The machine keeps any overpaid money. The machine has 5 states. The four states 0c,
Figure 23.1: 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 label B means that the “Money back” button was pressed.

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 triangle on the left side of the circle. Figure 23.1 shows a diagram of the automaton COFFEE. 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 23.1**

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 several applications:

- 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 Chapter 32 in “Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein.
Finite automata can be used as a tokenizer to preprocess the source code of a computer program.

Basically, a finite automaton is a deterministic 1-tape Turing machine that is highly restricted:

- The Turing machine $M$ cannot change symbols on the tape.
- It can only move the head to the right.\footnote{This renders the first condition useless, but we keep the first one for aesthetic reasons.}

In other words, $M$ has no memory and can read each symbol of the input once and that’s it. “No memory” means no memory on the tape, $M$ can store a finite amount of information in its states. This is—surprise!—the reason why $M$ is called a finite automaton.

After throwing away all the things that are not necessary, this is what is left of the Turing machine:

**Definition 23.1** A finite automaton is described by a 5-tuple $(Q, \Sigma, \delta, q_0, Q_{\text{acc}})$:

1. $Q$ is the set of states.
2. $\Sigma$ is the input alphabet.
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
4. $q_0 \in Q$ is the start state.
5. $Q_{\text{acc}} \subseteq Q$ is the set of accepting states.

### 23.2.1 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 $w = w_1 \ldots w_n$ 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. We already defined configurations and computations for Turing machines. We could use this definition for finite automata, too. But the only thing of a configuration that remains relevant for a finite automaton is the state. So a computation is essentially just a sequence of states. To get a clear presentation, let us make this explicit:

**Definition 23.2** Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a finite automaton. Let $w \in \Sigma^*$.

1. A sequence $s_0, \ldots, s_n$ 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 23.3  
1. Since $\delta$ 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 23.5 below, we show that we can always assume that $\delta$ is total and that there are no unfinished computations.

Definition 23.4  
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. $\text{REG}$ denotes the set of all regular languages.$^3$

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, and 12B12B12B22 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 $\Sigma$. ($\delta^* \text{ will play the role of } \vdash^*\text{.)}$ We inductively define the extended transition function $\delta^* : Q \times \Sigma^* \to Q$. We first defined $\delta^*$ for words of length 0, then of length 1, length 2, and so on. For $q \in Q$, $w \in \Sigma^*$, and $\sigma \in \Sigma$, we

---

$^3$For $\text{REG}$, we are careful and do not fix the alphabet size to $\{0, 1\}$. For $\text{REC}$, working solely over $\{0, 1\}$ was no problem, since one can encode larger alphabets. Finite automata are weak and might not be able to decode the encoding. Therefore, the definition of $\text{REG}$ depends on the alphabet $\Sigma$. There are two solutions: Either instead of $\text{REG}$ we define for each $\Sigma$ the class $\text{REG}_{\Sigma}$ of regular languages over $\Sigma$. Or we take a large $\Sigma_0$, like all symbols occuring in this script. Then the input alphabet of every finite automaton in this script is a subset of $\Sigma_0$ and we can consider this automaton as an automaton over $\Sigma_0$. Any solution is fine.

$^4$Not all customers are smart . . .
Finite automata and regular languages

Figure 23.2: The finite automaton $M_1$. The double circle around state 3 indicates that 3 is an accepting state.

set

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

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, $\delta$ and $\delta^*$ coincide, which is what we want. Intuitively, $\delta^*(q, w)$ is the state that the automaton reaches if it starts in state $q$ and then reads $w$.

23.2.2 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 not be 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 much to look at all of them at once and too much 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.

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 23.2 The automaton $M_1$ basically searches for the string $000$ in the word $w$. Design a similar automaton that searches for the sequence $01011$. Can you devise an algorithm that given any sequence $s$, constructs an automaton that searches for $s$ in a given word $w$?

23.3 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 $L \in \text{REG}$ is $\overline{L} = \Sigma^* \setminus L$ 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 $\overline{M}$ for $\overline{L}$. 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 $\overline{M}$. If $w \in L(M)$, then $\overline{M}$ will indeed reject $w$, since the accepting computation of $M$ on $w$ is turned into a rejecting one of $\overline{M}$. If $w \notin L(M)$, there is a problem: A rejecting computation of $M$ on $w$ is turned into an accepting computation of $\overline{M}$ 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 $\overline{M}$ on $w$ will also be unfinished and therefore $w \notin L(\overline{M})$. 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 23.5 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 $\delta'$ is a total function and $L(M) = L(M')$.

Proof overview: We add an extra dead-lock state to $Q$. Whenever $\delta$ 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

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

$$
\delta'(\text{dead-lock}, \sigma) = \text{dead-lock}.
$$

$\delta'$ 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

\(^5\)We always assume that the universe $\Sigma^*$ of $L$ is known, so it is clear what the complement is.
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 $\text{REG}$ is closed under intersection and union, i.e, if $A, B \subseteq \Sigma^*$ are regular languages, so are $A \cup B$ and $A \cap B$. The construction that we will use is called \textit{product automaton}. We already defined the product of Turing machines. The product construction for finite automata is essentially the same, it is just simpler because finite automata are simpler. The product construction also works for the set difference $A \setminus B$. This in particular implies that $\text{REG}$ is closed under complementation, since $\Sigma^*$ is regular. 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 23.6** 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 $\delta_1$ and $\delta_2$ are total functions. Then the transition function $\Delta$ defined by

$$
\Delta : (Q_1 \times Q_2) \times \Sigma \rightarrow Q_1 \times Q_2
$$

$$
((q_1, q_2), \sigma) \mapsto (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))
$$

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

$$
\Delta^*((q_1, q_2), w) = \Delta(\Delta^*((q_1, q_2), w'), \sigma)
$$

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

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

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

The second equality follows from the induction hypothesis. ■

**Theorem 23.7** $\text{REG}$ 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
\( \delta_1 \) and \( \delta_2 \) are total functions. Let \( M = (Q_1 \times Q_2, \Sigma, \Delta, (q_{0,1}, q_{0,2}), F) \) where \( \Delta \) is the function as defined in Lemma 23.6.

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 23.6, \( \Delta^*((q_{0,1}, q_{0,2}), w) = (\delta^*(q_{0,1}, w), \delta^*(q_{0,2}, w)) \). 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 \( F = Q_{\text{acc,1}} \times (Q_2 \setminus Q_{\text{acc,2}}) \). \( \blacksquare \)
24 Nondeterministic finite automata

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

**Definition 24.1** 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

\[
A^* = \{x_1x_2 \ldots x_m \mid m \geq 0 \text{ and } x_\mu \in A, 1 \leq \mu \leq m\}
\]

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 \( A \).

**Example 24.2**

1. \( \{aa, ab\}\{a, bb\} = \{aaa, aabb, aba, abbb\} \).

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

**Exercise 24.1** Prove the following:

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

2. \( \emptyset^* = \{\varepsilon\} \) and \( \{\varepsilon\}^* = \{\varepsilon\} \).

**Exercise 24.2** 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.
24.1 Nondeterminism

When we showed that \( \text{REG} \) is closed under union or intersection, we took two automata for \( A, B \in \text{REG} \) 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 24.1 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. So this automaton is nondeterministic. We introduced nondeterminism in the context of Turing machines and the concept is the same for automata (which are restricted Turing machines): The automaton can choose which of the transitions it will make. 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 automaton for \( AB \). The amazing thing is, that, in contrast to general Turing machines, we know how to simulate a nondeterministic finite automaton by a deterministic one (without any time loss).

Another way to introduce nondeterminism are \( \varepsilon \)-transitions. These are arrows in the transition diagram that are labeled with \( \varepsilon \). This means that the automaton may choose to make the \( \varepsilon \)-transition without reading a symbol of the input. Figure 24.2 shows an automaton with \( \varepsilon \)-transitions for \( AB \).

24.1.1 Formal definition

Recall that 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 \( \Sigma \), \( \Sigma_\varepsilon \) denotes the set \( \Sigma \cup \{\varepsilon\} \).

**Definition 24.3** A nondeterministic finite automaton is described by a 5-tuple \( M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}}) \):

1. \( Q \) is the set of states.
2. \( \Sigma \) is the input alphabet.
3. \( \delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q) \) is the transition function.

© Markus Bläser 2007–2015
Figure 24.1: A nondeterministic finite automaton for $AB$.

Figure 24.2: A nondeterministic finite automaton with $\varepsilon$-transitions for $AB$.

The silver rule of nondeterminism

*Always remember the golden rule of nondeterminism.*

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

4. \( q_0 \in Q \) is the start state.

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

If \( \delta \) is only a function \( Q \times \Sigma \rightarrow \mathcal{P}(Q) \), then \( M \) is called a nondeterministic finite automaton without \( \varepsilon \)-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 \( \delta : Q \times \Sigma \rightarrow Q \) of a deterministic finite automata, we get a transition function \( Q \times \Sigma \rightarrow \mathcal{P}(Q) \) by \( (q, \sigma) \mapsto \{\delta(q, \sigma)\} \) for all \( q \in Q, \sigma \in \Sigma \). This means that the transition function does not map to states but to sets consisting of a single state.

24.1.2 Computations and computation trees

A nondeterministic finite automaton starts in \( q_0 \). Then it has several choices. There is the possibility to make an \( \varepsilon \)-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 24.4** Let \( M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}}) \) be nondeterministic finite automaton. Let \( w \in \Sigma^* \), \( w = w_1 \ldots w_n \).

1. \( s_0, s_1, \ldots, s_m \in Q \) is called a computation of \( M \) on \( w \) if we can write \( w = u_1 u_2 \ldots u_m \) with \( u_\mu \in \Sigma_\varepsilon \) such that
   
   (a) \( s_0 = q_0 \).
   
   (b) For all \( 0 \leq \mu < m \), \( s_{\mu+1} \in \delta(s_\mu, u_{\mu+1}) \).

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

**Definition 24.5**

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 \( M \).

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 \( \delta \) as we did for deterministic finite automata. But this is not as easy. We first define the \( \varepsilon \)-closure of the transition function. For \( q \in Q \)
and \( \sigma \in \Sigma \), \( \delta^{(e)}(q, \sigma) \) denotes all states that we can reach from \( q \) by making an arbitrary number of \( \varepsilon \)-transitions and then one transition that is labeled with \( \sigma \). (And we are not allowed to make any \( \varepsilon \)-transitions afterwards.) Formally,

\[
\delta^{(e)} : Q \times \Sigma \to \mathcal{P}(Q) \\
(q, \sigma) \mapsto \{ r \mid \text{there are } k \geq 0 \text{ and } s_0 = q, s_1, \ldots, s_k \text{ such that} \\
s_{k+1} \in \delta(s_k, \varepsilon), 0 \leq k < k, \text{ and } r \in \delta(s_k, \sigma) \}.
\]

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

\[
R^{(e)} = \{ r \in Q \mid \text{there are } k \geq 0 \text{ and } s_0 = r, s_1, \ldots s_k \text{ such that} \\
s_{k+1} \in \delta(s_k, \varepsilon), 0 \leq k < k, \text{ and } s_k \in R. \}.
\]

**Lemma 24.6** If \( M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}}) \) is a nondeterministic finite automaton, then \( M' = (Q, \Sigma, \delta^{(e)}, q_0, Q_{\text{acc}}^{(e)}) \) is a nondeterministic finite automaton without \( \varepsilon \)-transitions such that \( L(M) = L(M') \).

**Proof.** \( M' \) obviously does not have \( \varepsilon \)-transitions. Assume that \( M \) accepts \( w \). Then we can write \( w = u_1u_2 \ldots u_m \) with \( u_\nu \in \Sigma_{\varepsilon} \), and there are states \( s_0, s_1, \ldots, s_m \) with \( s_0 = q_0, \delta(s_\mu, u_\mu) = s_{\mu+1} \) for \( 0 \leq \mu < m \), and \( s_m \in Q_{\text{acc}} \).

Let \( i_1, \ldots, i_n \) be indices such that \( u_{i_\nu} \in \Sigma \) for \( 1 \leq \nu \leq n \), i.e., \( u_{i_\nu} \) is the \( \nu \)th symbols of \( w \). Let \( i_0 = 0 \). Then \( \delta^{(e)}(s_{i_\nu}, u_{i_{\nu+1}}) = s_{i_{\nu+1}} \) for \( 0 \leq \nu < n \) and \( s_{i_n} \in Q_{\text{acc}}^{(e)} \) by the construction of \( \delta^{(e)} \) and \( Q_{\text{acc}}^{(e)} \). Thus \( M' \) accepts \( w \), too.

Conversely, if \( M' \) accepts \( w = w_1w_2 \ldots w_n \) with \( w_\nu \in \Sigma \) for \( 1 \leq \nu \leq n \), then \( M \) also accepts \( w \). There are states \( s_0, \ldots, s_n \) such that \( s_0 = q_0 \) and \( \delta^{(e)}(s_{i_\nu}, w_{i_{\nu+1}}) = s_{i_{\nu+1}} \) for \( 0 \leq \nu < n \), and \( s_n \in Q_{\text{acc}}^{(e)} \). For all \( 0 \leq \nu < n \), there are states \( s_{i_\nu} = s_{i_{\nu}}, s_{i_{\nu+1}}, \ldots, s_{i_{\nu}+j_\nu} \) such that \( \delta(s_{i_{\nu}+h}, \varepsilon) = s_{i_{\nu}+h+1}, 0 \leq h < j_\nu \), and \( \delta(s_{i_{\nu}+j_\nu}, w_{i_{\nu+1}}) = s_{i_{\nu+1}} \) by the construction of \( \delta^{(e)} \). And there are states \( s_{i_n} = s_n, s_{i_{n+1}}, \ldots, s_{i_{n}+j_n} \) such that \( \delta(s_{i_{n}+h}, \varepsilon) = s_{i_{n}+h+1}, 0 \leq h < j_n \), and \( s_{i_{n}+j_n} \in Q_{\text{acc}}^{(e)} \) by the construction of \( Q_{\text{acc}}^{(e)} \). All these intermediate states together form an accepting computation of \( M \) on \( w \). \( \blacksquare \)

**Remark 24.7** All finished computations of a finite automaton without \( \varepsilon \)-transitions on a given input have the same length.

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:

\[
\delta^{*}(q, \varepsilon) = \{ q \} \quad \text{for all } q \in Q,
\]

\[
\delta^{*}(q, w\sigma) = \bigcup_{r \in \delta^{*}(q, w)} \delta^{(e)}(r, \sigma) \quad \text{for all } q \in Q, \ w \in \Sigma^{*}, \ \sigma \in \Sigma.
\]

\( \delta^{*}(q, x) \) are all the states that we can reach if we start from \( q \), read \( x \), and do not allow any \( \varepsilon \)-transition after reading the last symbol of \( x \).
24.1. Nondeterminism

As in the case of general Turing machines, it is sometimes more easier to represent the computations of a nondeterministic finite automaton $M$ by a computation tree. We only do this for $\varepsilon$-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. 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 $i$. $M$ 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 24.3 shows the computation tree of the automaton from Figure 24.1 on the word 01011.
24.2 Determinism versus nondeterminism

**Theorem 24.8** 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 24.6, we can assume that $M$ does not have any $\varepsilon$-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 $n$th level contains a state from $Q_{\text{acc}}$, 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 $P(Q)$, the power set of $Q$, as $\hat{Q}$, the set of states of $\hat{M}$.

*Proof.* By Lemma 24.6, we can assume that $M$ does not have any $\varepsilon$-transitions. We set $\hat{Q} = P(Q)$ and $\hat{q}_0 = \{q_0\}$. We define the transition function $\hat{\delta}$ as follows:

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

(Note that $\delta$ is a function $\hat{Q} \times \Sigma \rightarrow \hat{Q}$. It does map 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 $w = \epsilon$, $\hat{\delta}^*(R, w) = R = \delta^*(R, w)$. Now let $w = x\sigma$ with $\sigma \in \Sigma$. We have

$$\hat{\delta}^*(R, w) = \hat{\delta}(\hat{\delta}^*(R, x), \sigma)$$

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

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

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

Above, the second equality follows from the induction hypothesis and the third equality follows from the definition of $\hat{\delta}$. $\hat{M}$ accepts $w$ iff $\hat{\delta}^*(q_0, w) \in \hat{Q}_{\text{acc}}$. $M$ accepts $w$ iff $\delta^*(\{q_0\}, w) \cap Q_{\text{acc}} \neq \emptyset$. From the definition of $\hat{Q}_{\text{acc}}$ it follows that $M$ accepts $w$ iff $M$ accepts $w$. Thus $L(M) = L(M')$. \hfill \blacksquare
Exercise 24.3 Why does this approach not work for general nondeterministic Turing machines?

 Somehow one feels cheated when seeing the subset construction, 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 has $2^n$ states and there are examples where this is (almost) necessary. Here is one such example.

Example 24.9 For $n \in \mathbb{N}$, consider the language

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

There is a nondeterministic finite automaton for $S_n$ that has $n + 1$ states, see Figure 24.4 for $n = 3$. A deterministic finite automaton could store the 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 $\delta(\sigma_1 \ldots \sigma_n, \tau) = \sigma_2 \ldots \sigma_n \tau$ for all $\sigma_1 \ldots \sigma_n \in Q$ and $\tau \in \Sigma$. The start state is $1^n$ and the accepting states are all states of the form $0\sigma_2 \ldots \sigma_n$. This automaton has $2^n$ states. We will see soon that this is necessary.

24.3 Closure properties, part II

Theorem 24.10 REG is closed under concatenation and Kleene closure, i.e., for all $A, B \in \text{REG}$, we have $AB, A^* \in \text{REG}$.

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 $\varepsilon$-transition to the start state $q'_0$ of
M'. Formally, \[
\gamma_1(q, \sigma) = \begin{cases} 
\delta(q, \sigma) & \text{if } q \in Q \text{ and } \sigma \neq \varepsilon, \\
\delta'(q, \sigma) & \text{if } q \in Q' \text{ and } \sigma \neq \varepsilon, \\
q_0' & \text{if } q \in Q_{\text{acc}} \text{ and } \sigma = \varepsilon, \\
\text{undef.} & \text{otherwise.}
\end{cases}
\]

It is clear from the construction that \(L(N_1) = AB\).

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 \(\varepsilon\)-transition from every accepting 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 \(\varepsilon\) to the language, one idea would be to make the start state an accepting state, too. While this adds \(\varepsilon\) to the language, this does not work (cf. Exercise 24.4). Instead, we add a new start state \(r\) that is an accepting state. This adds \(\varepsilon\) to the language. From \(r\), there is an \(\varepsilon\)-transition to the old start state \(q_0\) of \(M\). Formally, \[
\gamma_2(q, \sigma) = \begin{cases} 
\delta(q, \sigma) & \text{if } q \in Q \text{ and } \sigma \neq \varepsilon, \\
q_0 & \text{if } q \in \{r\} \cup Q_{\text{acc}} \text{ and } \sigma = \varepsilon, \\
\text{undef.} & \text{otherwise.}
\end{cases}
\]

It is clear from the construction that \(L(N_2) = A^*\).

**Exercise 24.4** 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 \(\varepsilon\)-transition from each accepting state to the start state and make the start state an accepting state, too.

**24.4 Further exercises**

**Exercise 24.5** For a word \(w = w_1w_2\ldots w_n \in \Sigma^*\), let \(w^{\text{rev}}\) denote the word \(w_nw_{n-1}\ldots w_1\). For a language \(L \subseteq \Sigma^*\), \(L^{\text{rev}} = \{w^{\text{rev}} \mid w \in L\}\). Show that if \(L\) is regular, so is \(L^{\text{rev}}\).
25 Regular expressions

25.1 Formal definition

25.1.1 Syntax

Definition 25.1 Let \( \Sigma \) be an alphabet and assume that the symbols “(”, “)”, \( \emptyset \), “\( \varepsilon \)”, “+”, and “\( \ast \)” do not belong to \( \Sigma \). Regular expressions over \( \Sigma \) are defined inductively:

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

Above \( \emptyset \) and \( \varepsilon \) 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 \( \emptyset \) and \( \varepsilon \) instead of \( \varepsilon \). It is usually clear from the context whether we mean the symbols for the empty set and the empty word or the objects themselves.

25.1.2 Semantics

Definition 25.2 Let \( E \) be a regular expression. The language \( L(E) \) denoted 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 \( E = (E_1^\ast) \), then \( L(E) = L(E_1)^\ast \).

The symbol \( \emptyset \) represents the empty set and the symbol \( \varepsilon \) 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.

**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.

- $[\tau_1 \tau_2 \ldots \tau_\ell]$ is the union of the $k$ symbols, i.e, in our notation it is $\tau_1 + \tau_2 + \cdots + \tau_\ell$.

- 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 $EE \cdots E$, $m$ times.

For instance, elements of programming languages, so-called tokens are usually elements from a regular language. Consider our language WHILE and assume that we have to write it as an ASCII text. So *while* is not a symbol anymore but it is composed of the five letters while. The regular expression $[Ww][Hh][Ii][Ll][Ee]$ describes all possibilities to write the keyword *while*. We assume that WHILE is not a case-sensitive language. The expression $x0|x[1-9][0-9]^*$ describes the variable names that we are allowed to use. (We forbid leading zeros here.)

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 occurrences 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.).

The tools *grep* or *egrep* work with regular expressions, too. For instance, *egrep* `x0|x[1-9][0-9]*` *while.prg* gives you all occurrences of variables in the source code *while.prg*.

---

1But notice that the languages of all valid source codes are usually not regular. They are usually context-free, a concept that we will meet after Christmas. The use of “usually context-free” is questionable here, in particular, Bodo disagrees. The set of source codes of a “pure” and “simple” programming language like WHILE is context-free, the set of an overloaded one like C++ or JAVA usually is not.
25.2 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 25.3** 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. $\varepsilon E = E\varepsilon = E$ ($\varepsilon$ 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 part. The rest is left as an exercise.

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

Part 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.

Part 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 $x_\nu$ as $y_{\nu,1} \cdots y_{\nu,j_\nu}$ with $y_{\nu,i} \in L$ for $1 \leq i \leq j_\nu$, $1 \leq \nu \leq n$. This means that we can write $x_1 \cdots x_n = y_{1,1} \cdots y_{1,j_1} y_{2,1} \cdots y_{n-1,j_{n-1}} y_{n,1} \cdots y_{n,j_n} \in L^*$. ■

Exercise 25.1 1. Prove the remaining parts of Theorem 25.3.

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

Exercise 25.2 Prove the following identities:

1. $(E + F)^* = (E^* F^*)^*$.

2. $\varepsilon + EE^* = E^*$.

3. $(\varepsilon + E)^* = E^*$.

25.2.1 Precedence of operators

In the definition of regular expressions, whenever we built new expression out of two other, we placed brackets around the expression. Since these brackets often make the expressions hard to read, we are allowed to omit them. But then we have to make conventions about the order of precedence of operators and how to get back the completely parenthesized expressions.

- 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. We put brackets around this expression and the star.

- Next comes concatenation. All concatenations that are adjacent are grouped together. Since concatenation is associative, it does not matter in which order we group consecutive concatenations. But to get a deterministic procedure, we will always group them from the left to the right.

- Finally, we group all unions. Again, the order does not matter, but we will do it from the left to the right by convention.

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

Example 25.4 Consider the expression $01^*0 + 0 + \varepsilon$. It is transformed back into $(((0(1^*))0) + 0) + \varepsilon$.
25.3 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 25.5** If \(E\) is a regular expression, then \(L(E) \in \text{REG}\).

**Proof overview:** The proof is done by structural induction. We first prove the statement of the theorem for the simple regular expressions (Definition 25.1, 1. and 2.). This is the induction basis. In the induction step, we have prove the statement for the expressions \(E_1 + E_2\), \(E_1 E_2\), 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 25.3).

**Induction step:** We are given \(E = E_1 + E_2\), \(E = E_1 E_2\), 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 \(\text{REG}\) is closed under union, concatenation and Kleene closure.

**Exercise 25.3** Construct finite automata that accept the languages \(\emptyset\), \(\{\varepsilon\}\), and \(\{\sigma\}\) for \(\sigma \in \Sigma\).

**Theorem 25.6** 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 \(j_1, \ldots, j_k\) 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 \(L(M) = L(E_{i,j_1}^n + \cdots + E_{i,j_k}^n)\).

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
\[
L(E^k_{i,j}) = \{ w \mid \text{ for each prefix } w' \text{ of } w \\
\qquad \text{with } w' \neq \varepsilon \text{ and } w' \neq w, \delta^*(i, w') \leq k \}
\] (25.1)
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 (25.1).

Induction step: Assume that we have found regular expressions such that (25.1) holds for some \(k\). We have to construct the expressions \(E^{k+1}_{i,j}\). A path that goes from \(i\) to \(j\) and all states in between are from \(\{1, \ldots, k+1\}\) can go from \(i\) to \(j\) with only going through states from \(\{1, \ldots, k\}\). For this, we know already an 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 \(E^k_{i,k+1}(E^k_{k+1,k+1})^*E^{k+1}_{k+1,j}\).

Altogether, we have \(E^{k+1}_{i,j} = E^k_{i,j} + E^k_{i,k+1}(E^k_{k+1,k+1})^*E^{k+1}_{k+1,j}\). It is clear that \(L(E^k_{i,j})\) is a subset of the language on the right-hand side of (25.1). 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 \(x_{i+1} = x_iy_i\). 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 \(x_sy_s = w\). But this means that \(w \in L(E^k_{i,j})\). This proves (25.1).

From (25.1), it follows that \(L(M) = L(E^n_{1,j_1} + \cdots + E^n_{1,j_t})\) where \(Q_{\text{acc}} = \{j_1, \ldots, j_t\}\). \(\blacksquare\)

Remark 25.7 The algorithm above does essentially the same as the Floyd–Warshall algorithm for computing all-pair shortest paths. If we replace \(E^{k+1}_{i,j} = E^k_{i,j} + E^k_{i,k+1}(E^k_{k+1,k+1})^*E^{k+1}_{k+1,j}\) by \(d^{k+1}_{i,j} = \min\{d^k_{i,j}, d^k_{i,k+1} + d^k_{k+1,j}\}\), we can compute the shortest distances between all pairs of nodes.
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? Well there are since there are undecidable languages, but are there simple non-regular languages? Here is one example:

$$A = \{0^n1^n \mid n \in \mathbb{N}\}.$$ 

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 in 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 $0^n1^n$—there is not, but you have to give a formal proof.)

26.1 The pumping lemma

**Lemma 26.1 (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 $i \in \mathbb{N}$, $uxy^i zw \in L$.

**Proof overview:** We choose $n$ to be the number of states of a finite automaton $M$ that recognizes $L$. 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 $uxy^i zw$.

Below, there is a automaton with $uvw$ on the input tape. In the second line, there are the states of $M$ when moving from one cell to the next. By the pigeon hole principle, two of them are equal, say, $q_i = q_j$ with $i < j$. The word $v_{i+1} \ldots v_j$ between them can be "pumped".

\[
\begin{array}{ccccccc}
\ldots & u_{n-1} & v_1 & v_2 & \ldots & v_n & w_1 & \ldots \\
q_0 & q_1 & q_2 & \ldots & q_{n-1} & q_n & \\
\end{array}
\]
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 \( v = yzw \) 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 \( s_{j_1} = s_{j_2} \),

\[
\delta^*(q_0, uxyzw) = \delta^*(s_{j_1}, yzw) = \delta^*(s_{j_1}, y^{i-1}zw) = \ldots = \delta^*(s_{j_1}, yzw) = \delta^*(s_{j_1}, zw) \in Q_{\text{acc}}
\]

Thus \( uxyzw \in L \) for all \( i \in \mathbb{N} \). □

### 26.2 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 26.2** Let us show that \( A = \{0^n1^n \mid n \in \mathbb{N} \} \) 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 \( uxy^izw \notin A \), 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 \( uxy^izw = 0^{n+(i-1)s}1^n \). 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 fulfill the condition of the pumping lemma and thus is not regular.

© Markus Bläser 2007–2015
Example 26.3 Closure properties are sometimes useful to simplify proofs of non-regularity: Consider

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

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 \( uxy^i zw \notin L \). Your professor wins if \( uxy^i zw \in L \). 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.

26.3 How to decide properties of regular languages

Not part of the lecture

Which properties can we decide about regular languages? Can we decide whether a regular language is empty? This question does not seem to be
meaningful since a regular language is empty if it is empty. So the true question is: Given a representation of a regular language, for instance a deterministic finite automaton $M$, is $L(M)$ empty? Here are the problems that we will consider:

**Word problem:** Given $M$ and $w \in \Sigma^*$, is $w \in L(M)$?

**Emptiness problem:** Given $M$, is $L(M) = \emptyset$?

**Finiteness problem:** Given $M$, is $|L(M)| < \infty$?

**Equivalence problem:** Given $M_1$ and $M_2$, is $L(M_1) = L(M_2)$?

The running time of the decision procedures depends on the representation of the regular language. We always assume that the regular language is given by a deterministic finite automaton. Of course, if we are given a nondeterministic finite automaton, we can transform it into an equivalent deterministic one, but this new automaton is much larger than the nondeterministic one. Nevertheless, some of the decision procedures that we give even work for nondeterministic automata without any changes.

### 26.3.1 The word problem

Testing whether $w \in L(M)$ is easy. We just compute $\delta^*(q_0, w)$ and test whether it is in $Q_{\text{acc}}$. This takes $|w|$ steps if the automaton is deterministic. If $M$ is nondeterministic, then we just compute a list of states that can be reached when reading a longer and longer prefix of $w$.

### 26.3.2 Testing emptiness

If a given deterministic finite automaton $M$ has $n$ states, then $L(M)$ contains a word of length $< n$, provided that $L(M)$ is non-empty. This follows from the pumping lemma. Assume we have a word $w \in L(M)$ with $|w| \geq n$. Then the pumping lemma says that there must be a shorter word in $L(M)$, namely the one that we get when we set $i = 0$. Thus in order to decide whether $L(M)$ is empty or not, we “just” have to check all words of length $< n$. While these are finitely many, there are still a lot, namely $\sum_{\nu=0}^{n-1} |\Sigma|^\nu = \frac{|\Sigma|^n - 1}{|\Sigma| - 1}$. A much faster algorithm is obtained by viewing the transition diagram as a directed graph and then checking whether there is a path from the start state to an accepting state.

### 26.3.3 Testing finiteness

$L(M)$ is infinite if $L(M)$ contains a word of length between $n$ and $2n - 1$ where $n$ is the number of states of $M$: If $L(M)$ is infinite, then it contains words that are arbitrarily long. By the pumping lemma, as long as the word
has length $\geq 2n$, 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 the language has a word with length between $n$ and $2n - 1$, then we can pump this word and thus $L(M)$ is infinite. Again we get a faster algorithm by using graphs algorithms to check whether a node in the transition diagram lies on any cycle and then checking whether there is a path from the start state to some accepting state that contains a node on a cycle.

### 26.3.4 Testing equivalence

When is $L(M_1) = L(M_2)$? This is equivalent to $L(M_1) \setminus L(M_2) = \emptyset$ and $L(M_2) \setminus L(M_1) = \emptyset$. Thus we just have to construct the appropriate product automata and can reduce the problem to emptiness testing.

#### 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 $u = \epsilon$, $v$ the prefix of length $n$ of $t$, and $w$ 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 unnecessarily complicated. We can just choose $u = 0^i1$, $v = 0^p$, and $w = 1^n$ and are done.

### 26.4 Further exercises

**Exercise 26.1** Show that the following languages are not regular by using the pumping lemma.

1. $L = \{0^n1^m \mid n > m\}$,
2. $L = \{0^p \mid p \text{ is prime}\}$,
3. $L = \{a^ib^jc^k \mid i + j = k\}$,
Exercise 26.2  Show that no infinite subset of $\{0^n1^n \mid n \in \mathbb{N}\}$ is regular.

Exercise 26.3  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 Myhill-Nerode theorem
and minimal automata

Recall that an equivalence relation is a relation that is reflexive, symmetric, and transitive. We here consider equivalence relations on $\Sigma^*$. 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 $\Sigma^*$, that means, they are pairwise disjoint and their union is $\Sigma^*$. We call a relation right invariant (with respect to concatenation) if for all $x, y \in \Sigma^*$,

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

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.)

**Definition E.1 (Automaton relation)** Let $M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}})$ be a deterministic finite automaton. The relation $\equiv_M$ is defined on $\Sigma^*$ by

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

**Lemma E.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 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 E.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 $\Sigma^*$ 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 \( \Sigma^* \) 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 \( \sim_{L(M)} \). The relation \( \sim_L \) is even defined for languages that are not regular.

**Definition E.4 (Myhill-Nerode relation)** Let \( L \subseteq \Sigma^* \). The Myhill-Nerode relation \( \sim_L \) is defined on \( \Sigma^* \) by

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

**Lemma E.5** For every \( L \subseteq \Sigma^* \), \( \sim_L \) is an equivalence relation that is right invariant.

**Proof.** \( \sim_L \) is an equivalence relation, since \( \iff \) is an equivalence relation.

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 \( z' = wz \).

### E.1 The Myhill-Nerode theorem

**Theorem E.6 (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. \( \implies \) 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,

\[
L = \{ x \in \Sigma^* \mid \delta^*(q_0, x) \in Q_{\text{acc}} \} \\
= \bigcup_{x : \delta^*(q_0, x) \in Q_{\text{acc}}} \langle x \rangle_M.
\]

Although the union is over infinitely many words \( x \), only finitely many distinct equivalence classes appear in it.

2. \( \implies \) 3.: Let \( R \) be a right invariant equivalence relation with finite index such that \( L = \bigcup_{i \geq 1} \langle x_i \rangle_R \). 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
E.1. The Myhill–Nerode theorem

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,

\[
xzRyz \quad \text{for all } z \in \Sigma^*.
\]

Since \( L = \bigcup \{ [x_1]_R \cdots [x_t]_R \} \), \( R \)-equivalent words are either both in \( L \) or both not in \( L \). Hence (E.1) implies

\[
xz \in L \iff yz \in L \quad \text{for all } z \in \Sigma^*.
\]

Thus \( x \sim_L y \).

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

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

\( \delta \) 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,

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

since the words in an equivalence classes of \( \sim_L \) are either all in \( L \) or all not in \( L \). □

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

Example E.7 Let \( A = \{ 0^n1^n \mid n \in \mathbb{N} \} \). We have \( 0^i \not\sim_A 0^j \) for \( i \neq j \), since \( 0^i1 \in A \) but \( 0^j1 \notin A \). 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.

### E.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 $\delta$ and $\delta'$ are total. $M$ and $M'$ are isomorphic if there is a bijection $i : Q \to Q'$ such that

1. for all $q \in Q$ and $\sigma \in \Sigma$,
   \[i(\delta(q, \sigma)) = \delta'(i(q), \sigma),\]
2. $i(q_0) = q'_0$, 
3. $i(Q_{\text{acc}}) = Q'_{\text{acc}}$.

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

Together with the second and third condition, this means that two isomorphic automata are the same up to renaming the states.

**Theorem E.8** Let $L \subseteq \Sigma^*$ be a regular language.

1. Any deterministic finite automaton $M' = (Q', \Sigma, \delta', q'_0, Q'_{\text{acc}})$ such that $\delta'$ 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 isomorphic to the automaton $M$ constructed in the “3. $\implies$ 1.”-part of the proof of the Myhill–Nerode theorem.
Proof. Part 1: If we combine the “1. \( \implies 2. \)” and the “2. \( \implies 3. \)”-part of the proof of the Myhill-Nerode theorem, we see that the relation \( \equiv_{M'} \) is a refinement of \( \sim_L \). Thus \( |Q'| \geq \text{index}(\equiv_{M'}) \geq \text{index}(\sim_L) \).

Part 2: Assume now that \( |Q'| = |Q| \). This means that \( \text{index}(\sim_L) = \text{index}(\equiv_{M'}) \). Since \( \equiv_{M'} \) is a refinement of \( \sim_L \), this means that 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

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

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 \( q \in Q \), \( \sigma \in \Sigma \), and
2. \( b(Q'_{\text{acc}}) = Q_{\text{acc}} \).

For the first statement, let \( q = [x] \), and let \( b^{-1}(q) = q' \). Then \( b(\delta'(q', \sigma)) = [x\sigma] \) by the definition of \( b \). For the second statement, let \( q' \in Q'_{\text{acc}} \) and let \( (\delta')^*(q_0, x) = q' \). Then \( b(q') = [x] \). \( x \in L(M') = L \) and thus \([x] \in Q_{\text{acc}}\). This argument can be reversed, and thus we are done. \( \blacksquare \)

Example E.9 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:

\[
\begin{align*}
    A_i &= \{ x \in \{0,1\}^* \mid \text{the number of } 1's \text{ in } x \text{ equals } i \} \quad \text{for } i = 0, 1, 2 \\
    A_{>2} &= \{ x \in \{0,1\}^* \mid \text{the number of } 1's \text{ in } x \text{ is } > 2 \}
\end{align*}
\]

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 \( > 2 \) 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. The corresponding minimal automaton is shown in Figure E.1.
This section was not treated in class and is not relevant for the exams.

E.3 An algorithm for minimizing deterministic finite automata

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 E.10 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 \( \{s_i, s'_j\} = \{s_j, s'_i\} \) 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 \( z' = z_1 \ldots z_{i} z_{i+1} \ldots z_{t} \). 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. \( \blacksquare \)

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. Basically, when we have such a \( z \) that proves that \( q \) and \( q' \) are not equivalent, then all the pairs we go through when reading \( z \) are not equivalent, too. Algorithm 1 constructs these pairs backwards, starting from these pairs that have exactly one state in \( Q_{\text{acc}} \) and one state not in \( Q_{\text{acc}} \).

**Algorithm 1** Minimization of deterministic finite automata

**Input:** a deterministic finite automaton \( M = (Q, \Sigma, \delta, q_0, Q_{\text{acc}}) \)

**Output:** the minimal deterministic finite automaton \( \hat{M} \) with \( L(M) = L(\hat{M}) \)

1. Initialize a Boolean table \( T \) of all unordered pairs from \( Q \) and set \( T(\{q, q'\}) = 1 \) for all pairs \( \{q, q'\} \).
2. Initialize a list \( L \) of unordered pairs from \( Q \).
3. For each pair \( \{q, q'\} \) such that \( q \in Q_{\text{acc}} \) and \( q' \notin Q_{\text{acc}} \) or vice versa, set \( T(\{q, q'\}) = 0 \) and add \( \{q, q'\} \) to \( L \).
4. while \( L \) is not empty do
   5. remove a pair \( \{q, q'\} \) from \( L \).
   6. for all pairs \( \{p, p'\} \) with \( \{\delta(p, \sigma), \delta(p', \sigma)\} = \{q, q'\} \) for some \( \sigma \in \Sigma \), if \( T(\{p, p'\}) = 1 \), then set \( T(\{p, p'\}) = 0 \) and add \( \{p, p'\} \) to \( L \).
7. while there is a pair \( \{q, q'\} \) with \( T(\{q, q'\}) = 1 \) do
   8. remove \( q \) from the set of states and set \( \delta(p, \sigma) = q' \) for all \( p \in Q \) and \( \sigma \in \Sigma \) with \( \delta(p, \sigma) = q \).
10. od
11. return the constructed automaton

**Exercise E.2**

1. Show that Algorithm 1 sets \( T(q, q') = 0 \) iff there is a \( z \) with \( \delta^*(q, z) \in Q_{\text{acc}} \) and \( \delta^*(q', z) \notin Q_{\text{acc}} \) or vice versa.

2. Show that Algorithm 1 constructs indeed the minimal deterministic finite automaton equivalent to \( M \).

**E.4 Further exercises**

**Exercise E.3** Let \( L_n = \{x \in \{0, 1\}^* \mid \text{the n-last symbol of } x \text{ is a } 1\} \).

1. Show that \( \text{index}(L_n) \geq 2^n \). Conclude that the power set construction that transforms a nondeterministic finite automaton into a deterministic one is essentially optimal.
2. Give a regular expression for $L_n$ that consists of a linear number of symbols.
27 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 27.1 A grammar $G$ is described by a 4-tuple $(V, \Sigma, P, S)$.

1. $V$ is a finite set, the set of variables or nonterminals.
2. $\Sigma$ 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 27.2 If $(u, v) \in P$ is a production, we will often write $u \rightarrow v$ 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 27.3 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 $v = x'y'z$ and there is a production $y \rightarrow y' \in P$. 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.)

¹Yes, I know, this is oversimplified and not complete.
Example 27.4 Let \( G_1 = (\{S\}, \{0, 1\}, P_1, S) \) where \( P_1 \) consists of the productions

\[
S \rightarrow \varepsilon \\
S \rightarrow 0S1
\]

Exercise 27.1 Prove by induction on \( i \) that \( 0^iS1^i \) is the only sentence of length \( 2i + 1 \) with \( S \Rightarrow^* G_1 0^iS1^i \). Conclude that \( L(G_1) = \{0^i1^i \mid i \in \mathbb{N}\} \).

Syntactic sugar 27.5 If \( P \) contains the productions \( u \rightarrow v_1, \ldots, u \rightarrow v_t \) (with the same left-hand sides \( u \)), then we will often write \( u \rightarrow v_1 | \cdots | v_t \).

Example 27.6 Let \( G_2 = (\{W, V, N, N^+, Z, Z^+\}, \{a, b, \ldots, z, 0, 1, \ldots, 9, ;, =, \neq, ;, +, -\}, P_2, W) \) where \( P_2 \) consists of the productions

\[
W \rightarrow V := V+V | V := V−V | V := N | \quad \text{while } V \neq 0 \text{ do } W \text{ od } | \\
W;W \\
V \rightarrow xN^+ \\
N^+ \rightarrow Z | Z^+N \\
N \rightarrow Z | ZN \\
Z \rightarrow 0 | 1 | \cdots | 9 \\
Z^+ \rightarrow 1 | 2 | \cdots | 9
\]

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 27.7 Let \( G_3 = (\{S, E, Z\}, \{0, 1, 2\}, P_3, S) \) where \( P_3 \) is given by

\[
S \rightarrow 0EZ | 0SEZ \\
ZE \rightarrow EZ \\
0E \rightarrow 01 \\
1E \rightarrow 11 \\
1Z \rightarrow 12 \\
2Z \rightarrow 22
\]

\(^2\)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.
27.1 The Chomsky hierachy

**Exercise 27.2**  
1. Prove by induction on \( n \) that \( S \xrightarrow{*} G_{3} 0^n1^n2^n \) for all \( n \geq 1 \).

2. Show that whenever \( S \xrightarrow{*} G_{3} w \), 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 \( 1Z \to 2 \) and there is an E to the right of the 2 created, then one cannot derive a word from \( \Sigma^* \) from the resulting sentence.

4. Conclude that \( L(G_{3}) = \{0^n1^n2^n \mid n \geq 1\} \).

### 27.1 The Chomsky hierarchy

**Definition 27.8** 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 \( \Sigma V \cup \Sigma \).

**Definition 27.9** 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 27.6 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 \( xAy \).

Type-3 grammars are also called right-linear grammars: “linear”, because the derivation trees (to be defined in the next chapter) degenerate essentially

---

3The 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.

© Markus Bläser 2007–2015
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 27.12 explains this: type-3 languages are exactly the regular languages. The grammar that we get from the grammar in Example 27.6 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 \(N \rightarrow ZN\) and \(N^+ \rightarrow Z^+ N\) by productions of the form \(N \rightarrow 0N\), \(N \rightarrow 1N\), etc.

**Definition 27.10**

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 27.4 is an “almost context-free” grammar for this language. The only problem is that we can derive \(\varepsilon\) 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

\[
S \rightarrow \varepsilon \mid S' \\
S' \rightarrow 01 \mid 0S'1
\]

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 \(A \rightarrow \varepsilon\) in an “almost context-free” grammar. (Note that this is not possible for context-sensitive grammars!)

**Syntactic sugar 27.11** 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 27.7, 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.

### 27.2 Type-3 languages

**Theorem 27.12** 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 $\epsilon \notin L(G)$. Let $F \notin V$. We set $M = (V \cup \{F\}, \Sigma, \delta, S, \{F\})$ where

$$
\delta(A, \sigma) = \begin{cases} 
\{B \mid A \rightarrow \sigma B \in P\} & \text{if } A \rightarrow \sigma \notin P \\
\{B \mid A \rightarrow \sigma B \in P\} \cup \{F\} & \text{otherwise.}
\end{cases}
$$

Let $w = w_1w_2\ldots w_n \in \Sigma^* \setminus \{\epsilon\}$. We have

$w \in L(G) \iff$ there are variables $V_1, \ldots, V_{n-1} \in V$ with $S \Rightarrow G w_1V_1 \Rightarrow G w_1w_2V_2 \Rightarrow G \ldots \Rightarrow G w_1w_2\ldots w_{n-1}V_{n-1} \Rightarrow G w_1w_2\ldots w_{n-1}w_n$

$\iff$ there are states $V_1, \ldots, V_{n-1} \in V$ with $V_1 \in \delta(S, w_1), V_2 \in \delta(V_1, w_2), \ldots, V_{n-1} \in \delta(V_{n-2}, w_{n-1}), F \in \delta(V_{n-1}, w_n)$

$\iff w \in L(M)$.

If $\epsilon \in L(G)$, then we first construct an automaton $M$ for $L(G) \setminus \{\epsilon\}$ first. Then we add a new start state that is also an end state and add an $\epsilon$-transition to the old start state of $M$. This automaton recognizes $L(M) \cup \{\epsilon\} = L(G)$.

$\Leftarrow$ is shown in Exercise 27.3. $lacksquare$

**Exercise 27.3** Let $M = (Q, \Sigma, \delta, q_0, Q_{acc})$ be a deterministic finite automaton. Let $G = (Q, \Sigma, P, q_0)$ where the set of productions contains a production $q \rightarrow \sigma q'$ for all $q, q' \in Q$ and $\sigma \in \Sigma$ with $\delta(q, \sigma) = q'$

and in addition

$q \rightarrow \sigma$ for all $q \in Q$, $q' \in Q_{acc}$, and $\sigma \in \Sigma$ with $\delta(q, \sigma) = q'$.

If $q_0 \in Q_{acc}$, we will add the production $q_0 \rightarrow \epsilon$, too. Show that $L(G) = L(M)$.

---

4This is possible, since we get a type-3 grammar for $L(G) \setminus \{\epsilon\}$ by removing the production $S \rightarrow \epsilon$. 

© Markus Bläser 2007–2015
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 27.4** 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 $V \Sigma \cup \Sigma$. Let $L \subseteq \Sigma^*$. Show that there is a left-linear grammar $G$ with $L(G) = L$ iff $L$ is regular.
28 Context-free grammars

By definition, context-free grammars have at most one $\varepsilon$-production, namely, $S \rightarrow \varepsilon$, 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 $\varepsilon$-productions. For instance, the grammar

$$S \rightarrow \varepsilon \mid 0S1$$

looks much nicer than the one given by

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow 01 \mid 0S1$$

For context-free grammars (but not for context-sensitive grammars!), we can allow arbitrary $\varepsilon$-productions. So from now on, we allow arbitrary $\varepsilon$-productions in context-free grammars. We will see that we can always find an equivalent grammar that has at most one $\varepsilon$-production of the form $S \rightarrow \varepsilon$.

28.1 Derivation trees and ambiguity

Consider the grammar $G$ given by

$$E \rightarrow E \ast E \mid E + E \mid (E) \mid x$$

It generates all arithmetic expressions with the operations $\ast$ and $+$ over the variable$^1$ $x$. A word $w \in \Sigma^*$ is in $L(G)$ if $S \Rightarrow^* w$. 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 the word $x + x \ast x$ in the example above:

$$E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E \ast E \Rightarrow x + x \ast x$$

$$E \Rightarrow E + E \Rightarrow E + E \ast E \Rightarrow E + E \ast x \Rightarrow E + x \ast x \Rightarrow x + x \ast x$$

$^1$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 $S \Rightarrow^*_G w$.

$^4$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 28.1** 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 \( \varepsilon \). 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 \( A \rightarrow x_1 x_2 \ldots x_t \in P \).

2. The yield (or leaf-word or front or ...) 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 28.1 shows the derivation tree that corresponds to the two derivations (28.1) and (28.2). The leftmost derivation (28.1) is obtained by doing a depth-first search and visiting the children from left to right, the rightmost derivation (28.2) 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 \( x + x \ast x \). It is shown in Figure 28.2. Having several derivation trees for the same word is in general a bad thing.

**Definition 28.2**

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 28.2 is unnatural, because it does not respect the usual precedence of the operators “\( \ast \)” 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, \( A \rightarrow \varepsilon \) is a production of \( P \).

6. There are far too many names for this.
28.1. Derivation trees and ambiguity

Figure 28.1: A derivation tree for $x + x * x$.

Figure 28.2: Another derivation tree for $x + x * x$.
unambiguous grammar:

\[
E \to T \mid T + E \\
T \to F \mid F \ast T \\
F \to x \mid (E)
\]

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 \( F \ast F \ast \cdots \ast F \) from \( T \). (This means that the root of the tree is labeled by \( T \).) In turn, there is only one derivation tree for deriving \( e_1 \ast e_2 \ast \cdots \ast e_t \) from \( T \) where each \( e_r \) 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 \( T \to F \mid T \ast T \), then the grammar still would be ambiguous.

There are context-free languages that are inherently ambiguous.

**Theorem 28.3 (without proof)** The language \( \{0^n1^m3^m \mid n,m \geq 1\} \cup \{0^n1^m2^m3^n \mid n,m \geq 1\} \) is context-free and inherently ambiguous.

**Exercise 28.1** Show that the language from Theorem 28.3 is context-free.

### 28.2 Elimination of useless variables

*This section was not treated in class and is not relevant for the exams.*

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 28.4** 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 28.5** 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 $S \Rightarrow^*_H uAv$. (Note that $S$ is still in $G'$, since $L(G) \neq \emptyset$.) But every variable in a corresponding 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 $w \in L(H)$. $\blacksquare$
Example 28.6  Consider the following grammar

\[
S \rightarrow AB \mid 0 \\
A \rightarrow 0
\]

We cannot derive any terminal word from \( B \), hence we remove the production \( S \rightarrow AB \). Now we cannot derive any sentence of the form \( xAy \) from \( S \), hence we remove the rule \( A \rightarrow 0 \), 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 \( S \rightarrow AB \) in the second step. The production \( A \rightarrow 0 \) would not have been removed.

Theorem 28.5 provides a way to eliminate useless symbols once we can determine the generating and reachable variables. Algorithms 2 and 3 solve these two tasks.

Exercise 28.2  Show that the Algorithms 2 and 3 are indeed correct.

Algorithm 2  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 \( A \rightarrow u \) with \( u \in \Sigma^* \).

2. while there is a production \( A \rightarrow a_1 a_2 \ldots a_t \) such that \( A \notin V_0 \) and all \( a_\tau \) that are in \( V \) are in \( V_0 \) do

3. Add \( A \) to \( V_0 \).

4. od

5. Return \( V_0 \).

Algorithm 3  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 \( S \rightarrow xAy \) for some \( x, y \in (V \cup \Sigma)^* \).

2. while there is a production \( A \rightarrow a_1 a_2 \ldots a_t \) such that \( A \in V_1 \) and some \( a_\tau \) 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 \( V_1 \).
29  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 \( \varepsilon \)-productions.

29.1  Elimination of \( \varepsilon \)-productions

Definition 29.1  Let \( G = (V, \Sigma, P, S) \) be a context-free grammar. \( A \in V \) is called nullable if \( A \Rightarrow^* G \varepsilon \).

Theorem 29.2  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 \( A \rightarrow a_1 a_2 \ldots a_k \) by all \( 2^k \) productions, one for each possibility to leave an \( a_i \lambda \) out where \( a_{i_1}, \ldots, a_{i_\ell} \) are all nullable variables among \( a_1, a_2, \ldots, a_k \).

2. Remove every \( \varepsilon \)-production. (This removes also an \( \varepsilon \)-production that we might have introduced in the first step when \( a_1, \ldots, a_k \) 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:

"\( \Rightarrow \)". The proof is by induction on the length of the derivation:

Induction base: If \( A \rightarrow u \in Q \), then \( u \neq \varepsilon \) by construction. There is a production \( A \rightarrow a_1 a_2 \ldots a_t \), each \( a_\tau \in V \cup \Sigma \), and indices \( j_1, \ldots, j_\ell \) such that the concatenation of all \( a_\tau \) with \( \tau \notin \{ j_1, \ldots, j_\ell \} \) is \( u \) and all \( a_\tau \) with \( \tau \in \{ j_1, \ldots, j_\ell \} \) 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 \( B \rightarrow y \in Q \) 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, \( A \Rightarrow^*_G xyz = u \).

"\( \Leftarrow \)". Is left as an exercise.

Exercise 29.1  Show the "\( \Leftarrow \)"-direction of the proof of Theorem 29.2.
Theorem 29.2 provides a way to eliminate $\varepsilon$-productions once we can determine the nullable variables. Algorithm 4 solves the latter task.

**Exercise 29.2** Show that Algorithm 4 is indeed correct.

**Algorithm 4** 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 $A \in V$ to $V_0$ for which there is a production $A \rightarrow \varepsilon \in P$.
2. while there is a production $A \rightarrow a_1a_2\ldots a_t$ such that $A$ is not in $V_0$ and all $a_\tau$ are nullable do
3. Add $A$ to $V_0$.
4. od
5. Return $V_0$.

### 29.2 Elimination of chain productions

**Definition 29.3** Let $G = (V, \Sigma, P, S)$ be a context-free grammar. A production of the form $A \rightarrow B$ with $A, B \in V$ is called a chain production.

Like $\varepsilon$-productions, chain productions are useful for getting compact grammars; $E \rightarrow T \mid E^*T$ is an example. On the other hand, like $\varepsilon$-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 $A \rightarrow A$. We build a directed graph $H = (V, E)$. There is an edge $(A, B) \in E$ if there is a chain rule $A \rightarrow B \in P$. (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 $B_\tau \rightarrow B_{\tau+1} \in P$, $1 \leq \tau < t$, and $B_t \rightarrow B_1 \in P$. But this means that the variables $B_1, \ldots, B_t$ are interchangeable. 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 29.3** 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 $B_1 \rightarrow B_1$. 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 29.4 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 $\geq 1$ 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 $A \to u_\lambda$. $\blacksquare$

29.3 The Chomsky normal form

Definition 29.5 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 29.6 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 $\varepsilon$-productions and chain rules. Thereafter, $L(G)$ does not
contain the empty word anymore. For every $\sigma \in \Sigma$, we introduce a new variable $T_\sigma$, add the new production $T_\sigma \to \sigma$, and replace every occurrence of $\sigma$ in the productions in P by $T_\sigma$ 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 production is either of the form $A \to \sigma$ or $A \to A_1 A_2 \ldots A_t$ where $A_1, A_2, \ldots, A_t$ are all variables and $t \geq 2$. Hence we are almost there except that the right-hand sides might have too many variables. We can overcome this problem by introducing new variables $C_2, \ldots, C_{t-1}$ and replacing the production $A \to A_1 A_2 \ldots A_t$ by

$$
A \to A_1 C_2 \\
C_2 \to A_2 C_3 \\
\vdots \\
C_{t-2} \to A_{t-2} C_{t-1} \\
C_{t-1} \to A_{t-1} A_t
$$

The resulting grammar $G'$ is obviously in Chomsky normal form. It is easy to see that $L(G') = L(G) \setminus \{\varepsilon\}$ (see Exercise 29.4).

**Exercise 29.4** Prove that the grammar $G'$ constructed in the proof of Theorem 29.6 indeed fulfills $L(G') = L(G) \setminus \{\varepsilon\}$ (and even $L(G') = L(G)$, since we assumed that $G$ does not contain any $\varepsilon$-productions).

**Exercise 29.5** 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?

### 29.4 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 29.6** 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.)