Methods and Theory of Automata and Languages

Frank Stephan

November 9, 2016

Automata Theory is the science of the treatment of languages (sets of words over a finite alphabet) from an algorithmic and theoretical viewpoint; there are also connections to the corresponding subsets of natural numbers. Automata, grammars and expressions have been found as the basic and natural concepts to deal with these notions and besides their base domain, they have also been applied to investigate certain types of mathematical structures like groups and semigroups. These lecture notes are the combined notes on which the lecture notes for CS4232 Theory of Computation and CS5236 Advanced Automata Theory are based. These nodes include general methods from automata theory and also various applications like automatic structures and the theory of games on finite graphs.

Frank Stephan: Rooms S17#07-04 and COM2#03-11
Departments of Mathematics and Computer Science
National University of Singapore
10 Lower Kent Ridge Road, Singapore 119076
Republic of Singapore
Telephone 6516-2759 and 6516-4246
Email fstephan@comp.nus.edu.sg
Homepage http://www.comp.nus.edu.sg/~fstephan/index.html

For further reading, the author refers to the textbooks *Automata Theory and its Applications* by Bakhadyr Khoussainov and Anil Nerode [46] and *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani and Ullman [35]. Epstein, Cannon, Holt, Levy, Paterson and Thurston [25] give background on the group-theoretic parts of this lecture.

The author would like to thank Volker Diekert, Henning Fernau, Vladimir Gusev, Sanjay Jain, Bakhadyr Khoussainov, Dietrich Kuske, Alexander Lauser, Wolfgang Merkle and Pavel Semukhin for correspondence.

Contents

1 Sets and Regular Expressions

In theoretical computer science, one considers several main ways to describe a language L; here a language is usually a set of strings w over an alphabet Σ. The alphabet Σ is usually finite. For example, $\{\varepsilon, 01, 10, 0011, 0101, 0110, 1001, 1010, 1100, 000111, \ldots\}$ is the language of all strings over $\{0, 1\}$ which contain as many 0 as 1. Furthermore, let vw or $v \cdot w$ denote the concatenation of the strings v and w by putting the symbols of the second string behind those of the first string: $001 \cdot 01 = 00101$. Sets of strings are quite important, here some ways to define sets.

Definition 1.1. (a) *A finite list in set brackets denotes the set of the corresponding elements, for example* $\{001, 0011, 00111\}$ *is the set of all strings which have two* 0*s followed by one to three* 1*s.*

(b) *For any set L, let L^* be the set of all strings obtained by concatenating finitely many strings from L:* $L^* = \{u_1 \cdot u_2 \cdot \ldots \cdot u_n : n \in \mathbb{N} \wedge u_1, u_2, \ldots, u_n \in L\}$.

(c) *For any two sets L and H, let $L \cup H$ denote the union of L and H, that is, the set of all strings which are in L or in H.*

(d) *For any two sets L and H, let $L \cap H$ denote the intersection of L and H, that is, the set of all strings which are in L and in H.*

(e) *For any two sets L and H, let $L \cdot H$ denote the set* $\{v \cdot w : v \in L \wedge w \in H\}$*, that is, the set of concatenations of members of L and H.*

(f) *For any two sets L and H, let $L - H$ denote the set difference of L and H, that is,* $L - H = \{u : u \in L \wedge u \notin H\}$.

Remarks 1.2. For finite sets, the following additional conventions are important: The symbol $\emptyset$ is a special symbol which denotes the empty set – it could also be written as $\{\,\}$. The symbol ε denotes the empty string and $\{\varepsilon\}$ is the set containing the empty string.

In general, sets of strings considered in this lecture are usually sets of strings over a fixed alphabet Σ. Σ^* is then the set of all strings over the alphabet Σ.

Besides this, one can also consider L^* for sets L which are not an alphabet but already a set of strings themselves: For example, $\{0, 01, 011, 0111\}^*$ is the set of all strings which are either empty or start with 0 and have never more than three consecutive 1s. The empty set $\emptyset$ and the set $\{\varepsilon\}$ are the only sets where the corresponding starred set is finite: $\emptyset^* = \{\varepsilon\}^* = \{\varepsilon\}$. The operation $L \mapsto L^*$ is called the "Kleene star operation" named after Stephen Cole Kleene who introduced this notion.

An example for a union is $\{0, 11\} \cup \{01, 11\} = \{0, 01, 11\}$ and for an intersection is $\{0, 11\} \cap \{01, 11\} = \{11\}$. Note that $L \cap H = L - (L - H)$ for all sets L and H.

Formal languages are languages L for which there is a mechanism to check membership in L or to generate all members of L. The various ways to describe a language

L are given by the following types of mechanisms:

- By a mechanism which checks whether a given word w belongs to L. Such a mechanism is called an automaton or a machine.
- By a mechanism which generates all the words w belonging to L. This mechanism is step-wise and consists of rules which can be applied to derive the word in question. Such a mechanism is called a grammar.
- By a function which translates words to words such that L is the image of another (simpler) language H under this function. There are various types of functions f to be considered and some of the mechanisms to compute f are called transducers.
- An expression which describes in a short-hand the language considered like, for example, $\{01, 10, 11\}^*$. Important are here in particular the regular expressions.

Regular languages are those languages which can be defined using regular expressions. Later, various characterisations will be given for these languages. Regular expressions are a quite convenient method to describe sets.

Definition 1.3. *A regular expression denotes either a finite set (by listing its elements), the empty set by using the symbol $\emptyset$ or is formed from other regular expressions by the operations given in Definition 1.1 (which are Kleene star, concatenation, union, intersection and set difference).*

Convention. For regular expressions, one usually fixes a finite alphabet Σ first. Then all the finite sets listed are sets of finite strings over Σ. Furthermore, one does not use complement or intersection, as these operations can be defined using the other operations. Furthermore, for a single word w, one writes a^* in place of $\{a\}^*$ and abc^* in place of $\{ab\} \cdot \{c\}^*$. For a single variable w, w^* denotes $(w)^*$, even if w has several symbols. L^+ denotes the set of all non-empty concatenations over members of L; so L^+ contains ε iff L contains ε and L^+ contains a non-empty string w iff $w \in L^*$. Note that $L^+ = L \cdot L^*$. Sometimes, in regular expressions, $L + H$ is written in place of $L \cup H$. This stems from the time where typesetting was mainly done only using the symbols on the keyboard and then the addition-symbol was a convenient replacement for the union.

Example 1.4. The regular language $\{00, 11\}^*$ consists of all strings of even length where each symbol in an even position (position 0, 2, ...) is repeated in the next odd position. So the language contains 0011 and 110011001111 but not 0110.

The regular language $\{0,1\}^* \cdot 001 \cdot \{0,1,2\}^*$ is the set of all strings where after some 0s and 1s the substring 001 occurs, followed by an arbitrary number of 0s and 1s and 2s.

The regular set $\{00,01,10,11\}^* \cap \{000,001,010,010,100,101,110,111\}^*$ consists of all binary strings whose length is a multiple of 6.

The regular set $\{0\} \cup \{1,2,3,4,5,6,7,8,9\} \cdot \{0,1,2,3,4,5,6,7,8,9\}^*$ consists of all decimal representations of natural numbers without leading 0s.

Exercise 1.5. *List all members of the following sets:*

(a) $\{0,1\} \cdot \{0,1,2\}$;
(b) $\{0,00,000\} \cap \{00,000,0000\}$;
(c) $\{1,11\} \cdot \{\varepsilon,0,00,000\}$;
(d) $\{0,00\} \cdot \{\varepsilon,0,00,000\}$;
(e) $\{0,1,2\} \cdot \{1,2,3\} \cap \{1,2,3\} \cdot \{0,1,2\}$;
(f) $\{00,11\} \cdot \{000,111\} \cap \{000,111\} \cdot \{00,11\}$;
(g) $\{0,1,2\} \cup \{2,3,4\} \cup \{1,2,3\}$;
(h) $\{000,111\}^* \cap \{0,1\} \cdot \{0,1\} \cdot \{0,1\}$.

Exercise 1.6. *Assume A has 3 and B has 2 elements. How many elements do the following sets have at least and at most; it depends on the actual choice which of the bounds is realised: $A \cup B$, $A \cap B$, $A \cdot B$, $A - B$, $A^* \cap B^*$.*

Exercise 1.7. *Let A, B be finite sets and $|A|$ be the number of elements of A. Is the following formula correct:*

$$|A \cup B| + |A \cap B| = |A| + |B|?$$

Prove the answer.

Exercise 1.8. *Make a regular expression for $0^*1^*0^*1^* \cap (11)^*(00)^*(11)^*(00)^*$ which does not use intersections or set difference.*

Theorem 1.9: Lyndon and Schützenberger [49]**.** *If two words v, w satisfy $vw = wv$ then there is a word u such that $v, w \in u^*$. If a language L contains only words v, w with $vw = wv$ then $L \subseteq u^*$ for some u.*

Proof. If $v = \varepsilon$ or $w = \varepsilon$ then $u = vw$ satisfies the condition. So assume that v, w both have a positive length and are different. This implies that one of them, say w, is strictly longer than the other one. Let k be the greatest common divisor of the lengths $|v|$ and $|w|$; then there are i,j such that $v = u_1u_2\ldots u_i$ and $w = u_1u_2\ldots u_j$ for some words $u_1, u_2, \ldots, u_j$ of length k. It follows from $vw = wv$ that $v^jw^i = w^iv^j$. Now $|v^j| = |w^i| = ijk$ and therefore $v^j = w^i$. The numbers i,j have the greatest common divisor 1, as otherwise k would not be the greatest common divisor of $|v|$ and $|w|$. Thus the equation $v^j = w^i$ implies that for each $h \in \{1,\ldots,j\}$ there is some position where u_h is in one word and u_1 in the other word so that all u_h are equal to u_1.

This fact follows from the Chinese Remainder Theorem: For every possible combination (i',j') of numbers in $\{1,2,\ldots,i\} \times \{1,2,\ldots,j\}$ there is a position $h' \cdot k$ such that h' by i has remainder $i'-1$ and h' by j has remainder $j'-1$, that is, the parts of the upper and lower words at positions $h' \cdot k, \ldots, (h'+1) \cdot k - 1$ are $u_{i'}$ and $u_{j'}$, respectively. It follows that $v, w \in u_1^*$. Here an example for the last step of the proof with $i = 3$ and $j = 4$:

$$
\begin{aligned}
&u_1\,u_2\,u_3\,u_1\,u_2\,u_3\,u_1\,u_2\,u_3\,u_1\,u_2\,u_3,\\
&u_1\,u_2\,u_3\,u_4\,u_1\,u_2\,u_3\,u_4\,u_1\,u_2\,u_3\,u_4.
\end{aligned}
$$

The upper and the lower word are the same and one sees that each u_1 in the upper word is matched with a different u_h in the lower word and that all u_h in the lower word are matched at one position with u_h.

For the second statement, consider any language L such that all words $v, w \in L$ satisfy $vw = wv$. Let $v \in L$ be any non-empty word and u be the shortest prefix of v with $v \in u^*$. Now let w be any other word in L. As $vw = wv$ there is a word $\tilde{u}$ with $v, w \in \tilde{u}^*$. Now $u, \tilde{u}$ satisfy that their length divides the length of v and $|u| \leq |\tilde{u}|$ by choice of u. If $u = \tilde{u}$ then $w \in u^*$. If $u \neq \tilde{u}$ then one considers the prefix $\hat{u}$ of $u, \tilde{u}$ whose length is the greatest common divisor of $|u|, |\tilde{u}|$. Now again one can prove that $u, \tilde{u}$ are both in $\hat{u}^*$ and by the choice of u, $\hat{u} = u$: The words $u^{|\tilde{u}|}$ and $\tilde{u}^{|u|}$ are the same and the prefix $\hat{u}$ of u is matched with all positions in $\tilde{u}$ starting from a multiple of $|\hat{u}|$ so that $u \in \hat{u}^*$; similarly $\tilde{u} \in \hat{u}^*$. Thus $w \in u^*$. It follows that L is a subset of u^*. The case that L does not contain a non-empty word is similar: then L is either empty or $\{\varepsilon\}$ and in both cases the subset of the set u^* for any given u. ∎

Theorem 1.10: Structural Induction. *Assume that P is a property of languages such that the following statements hold:*

- *Every finite set of words satisfies P;*
- *If L, H satisfy P so do $L \cup H$, $L \cdot H$ and L^*.*

Then every regular set satisfies P.

Proof. Recall that words include the empty word ε and that finite sets can also be empty, that is, not contain any element.

The proof uses that every regular set can be represented by a regular expression which combines some listings of finite sets (including the empty set) by applying the operations of union, concatenation and Kleene star. These expressions can be written down as words over an alphabet containing the base alphabet of the corresponding regular language and the special symbols comma, opening and closing set bracket, normal brackets for giving priority, empty-set-symbol, union-symbol, concatenation-symbol and symbol for Kleene star. Without loss of generality, the normal brackets are used in quite redundant form such that every regular expression σ is either a listing of a finite set or of one of the forms $(\tau \cup \rho)$, $(\tau \cdot \rho)$, τ^* for some other regular expressions τ, ρ. In the following, for a regular expression σ, let $L(\sigma)$ be the regular language described by the expression σ.

Assume by way of contradiction that some regular set does not satisfy P. Now there is a smallest number n such that for some regular expression σ of length n, the set $L(\sigma)$ does not satisfy P. Now one considers the following possibility of what type of expression σ can be:

- If σ is the string $\emptyset$ then $L(\sigma)$ is the empty set and thus $L(\sigma)$ satisfies P;
- If σ lists a finite set then again $L(\sigma)$ satisfies P by assumption;
- If σ is $(\tau \cup \rho)$ for some regular expressions then τ, ρ are shorter than n and therefore $L(\tau), L(\rho)$ satisfy P and $L(\sigma) = L(\tau) \cup L(\rho)$ also satisfies P by assumption of the theorem;
- If σ is $(\tau \cdot \rho)$ for some regular expressions then τ, ρ are shorter than n and therefore $L(\tau), L(\rho)$ satisfy P and $L(\sigma) = L(\tau) \cdot L(\rho)$ also satisfies P by assumption of the theorem;
- If σ is τ^* for some regular expression τ then τ is shorter than n and therefore $L(\tau)$ satisfies P and $L(\sigma) = L(\tau)^*$ also satisfies P by assumption of the theorem.

Thus in all cases, the set $L(\sigma)$ is satisfying P and therefore it cannot happen that a regular language does not satisfy P. Thus structural induction is a valid method to prove that regular languages have certain properties. ∎

Remark 1.11. As finite sets can be written as the union of singleton sets and as every singleton set consisting of a word $a_1 a_2 \ldots a_n$ can be written as $\{a_1\} \cdot \{a_2\} \cdot \ldots \cdot \{a_n\}$, one can weaken the assumptions above as follows:

- The empty set and every set consisting of one word which is up to one letter long satisfies P;

- If L, H satisfy P so do $L \cup H$, $L \cdot H$ and L^*.

If these assumptions are satisfied then all regular sets satisfy the property P.

Definition 1.12. *A regular language L has polynomial growth iff there is a constant k such that at most n^k words in L are strictly shorter than n; a regular language L has exponential growth iff there are constants h, k such that, for all n, there are at least 2^n words shorter than $n \cdot k + h$ in L.*

Theorem 1.13. *Every regular language has either polynomial or exponential growth.*

Proof. The proof is done by structural induction over all regular sets formed by regular expressions using finite sets, union, concatenation and Kleene star. The property P for this structural induction is that a set has either polynomial growth or exponential growth and now the various steps of structural induction are shown.

First every finite set has polynomial growth; if the set has k members then there are at most n^k words in the set which are properly shorter than k. Note that the definition of "polynomial growth" says actually "at most polynomial growth" and thus the finite sets are included in this notion.

Now it will be shown that whenever L, H have either polynomial or exponential growth so do $L \cup H$, $L \cdot H$ and L^*.

Assume now that L, H have polynomial growth with bound functions n^i and n^j, respectively, with $i, j \geq 1$. Now $L \cup H$ and $L \cdot H$ have both growth bounded by n^{i+j}. For the union, one needs only to consider $n \geq 2$, as for $n = 1$ there is at most the one word ε strictly shorter than n in $L \cup H$ and $n^{i+j} = 1$. For $n \geq 2$, $n^{i+j} \geq 2 \cdot n^{\max\{i,j\}} \geq n^i + n^j$ and therefore the bound is satisfied for the union. For the concatenation, every element of $L \cdot H$ is of the form $v \cdot w$ where v is an element of L strictly shorter than n and w is an element of H strictly shorter than n; thus there are at most as many elements of $L \cdot H$ which are strictly shorter than n as there are pairs of (v, w) with $v \in L, w \in H, |v| < n, |w| < n$; hence there are at most $n^i \cdot n^j = n^{i+j}$ many such elements.

The following facts are easy to see: If one of L, H has exponential growth then so has $L \cup H$; If one of L, H has exponential growth and the other one is not empty then $L \cdot H$ has exponential growth. If L or H is empty then $L \cdot H$ is empty and has polynomial growth.

Now consider a language of the form L^*. If L contains words $v, w \in L$ with $vw \neq wv$ then $\{vw, wv\}^* \subseteq L^*$ and as $|vw| = |wv|$, this means that there are 2^n words of length $|vw| \cdot n$ in L for all $n > 0$; thus it follows that L has at least 2^n words of length shorter than $|vw| \cdot n + |vw| + 1$ for all n and L has exponential growth.

If L does not contain any words v, w with $vw \neq wv$, then by the Theorem 1.9 of Lyndon and Schützenberger, the set L is a subset of some set of the form u^* and thus

L^* is also a subset of u^*. Thus L^* has for each length at most one word and L^* has polynomial growth.

This completes the structural induction to show that all regular sets have either polynomial or exponential growth. ∎

Examples 1.14. The following languages have polynomial growth:

(a) $\{001001001\}^* \cdot \{001001001001\}^*$;
(b) $(\{001001001\}^* \cdot \{001001001001\}^*)^*$;
(c) $\{001001, 001001001\}^* \cdot \{0000, 00000, 000000\}^*$;
(d) $\emptyset \cdot \{00, 01, 10\}^*$;
(e) $\{0, 1, 00, 01, 10, 11, 000, 001, 010, 011\}$.

The following languages have exponential growth:

(f) $\{001, 0001\}^*$;
(g) $\{000, 111\}^* \cap \{0000, 1111\}^* \cap \{00000, 11111\}^*$.

As a quiz, check out the following related questions:

(a) Does $L \cap H$ have exponential growth whenever L, H do?
(b) Does $\{0101, 010101\}^*$ have exponential growth?
(c) Does $\{000, 001, 011, 111\}^* \cdot \{0000, 1111\}$ have exponential growth?
(d) Does the set $\{w : w \in \{0, 1\}^*$ and there are at most $\log(|w|)$ many 1s in $w\}$ have polynomial growth?
(e) Does the set $\{w : w \in \{0, 1\}^*$ and there are at most $\log(|w|)$ many 1s in $w\}$ have exponential growth?
(f) Is there a maximal k such that every set of polynomial growth has at most n^k members shorter than n for every n?

Proposition 1.15. *The following equality rules apply to any sets:*

(a) $L \cup L = L$, $L \cap L = L$, $(L^*)^* = L^*$, $(L^+)^+ = L^+$;
(b) $(L \cup H)^* = (L^* \cdot H^*)^*$ *and if* $\varepsilon \in L \cap H$ *then* $(L \cup H)^* = (L \cdot H)^*$;
(c) $(L \cup \{\varepsilon\})^* = L^*$, $\emptyset^* = \{\varepsilon\}$ *and* $\{\varepsilon\}^* = \{\varepsilon\}$;
(d) $L^+ = L \cdot L^* = L^* \cdot L$ *and* $L^* = L^+ \cup \{\varepsilon\}$;
(e) $(L \cup H) \cdot K = (L \cdot K) \cup (H \cdot K)$ *and* $K \cdot (L \cup H) = (K \cdot L) \cup (K \cdot H)$;
(f) $(L \cup H) \cap K = (L \cap K) \cup (H \cap K)$ *and* $(L \cap H) \cup K = (L \cup K) \cap (H \cup K)$;
(g) $(L \cup H) - K = (L - K) \cup (H - K)$ *and* $(L \cap H) - K = (L - K) \cap (H - K)$.

Proof. **(a)** $L \cup L$ consists of all words which appear in at least one of the copies of L, thus it equals in L. Similarly, $L \cap L = L$. $(L^*)^*$ consists of all words u of the form $w_1 w_2 \ldots w_n$ where $w_1, w_2, \ldots, w_n \in L^*$ and each w_m is of the form $v_{m,1} v_{m,2} \ldots v_{m,n_m}$ with $v_{m,1}, v_{m,2}, \ldots, v_{m,n_m} \in L$. Note that these concatenations can take ε in the case that $n = 0$ or $n_m = 0$, respectively. The word u is the concatenation of concatenations of words in L which can be summarised as one concatenation of words in L. Thus $u \in L^*$. For the other way round, note that $L^* \subseteq (L^*)^*$ by definition. If $\varepsilon \in L$ then $L^+ = L^*$ and $(L^+)^+ = (L^*)^*$ else $L^+ = L^* - \{\varepsilon\}$ and $(L^+)^+ = (L^* - \{\varepsilon\})^+ = (L^* - \{\varepsilon\})^* - \{\varepsilon\} = (L^*)^* - \{\varepsilon\} = L^* - \{\varepsilon\} = L^+$.

(b) $L^* \cdot H^*$ contains L and H as subsets, as one can take in the concatenation the first or second component from L, H and the other one as ε. Thus $(L \cup H)^* \subseteq (L^* \cdot H^*)^*$. On the other hand, one can argue similarly as in the proof of **(a)** that $(L^* \cdot H^*)^* \subseteq (L \cup H)^*$. In the case that $\varepsilon \in L \cap H$, it also holds that $L \cup H \subseteq L \cdot H$ and thus $(L \cup H)^* = (L \cdot H)^*$.

(c) It follows from the definitions that $L^* \subseteq (L \cup \{\varepsilon\})^* \subseteq (L^*)^*$. As **(a)** showed that $L^* = (L^*)^*$, it follows that all three sets in the chain of inequalities are the same and $L^* = (L \cup \{\varepsilon\})^*$. $\emptyset^*$ contains by definition ε as the empty concatenation of words from $\emptyset$ but no other word. The third equality $\{\varepsilon\}^* = \{\varepsilon\}$ follows from the first two.

(d) The equalities $L^+ = L \cdot L^* = L^* \cdot L$ and $L^* = L^+ \cup \{\varepsilon\}$ follow directly from the definition of L^+ as the set of non-empty concatenations of members of L and the definition of L^* as the set of possibly empty concatenations of members of L.

(e) A word u is in the set $(L \cup H) \cdot K$ iff there are words v, w with $u = vw$ such that $w \in K$ and $v \in L \cup H$. If $v \in L$ then $vw \in L \cdot K$ else $vw \in H \cdot K$. It then follows that $u \in (L \cdot K) \cup (H \cdot K)$. The reverse direction is similar. The equation $K \cdot (L \cup H) = (K \cdot L) \cup (K \cdot H)$ is be proven by almost identical lines of proof.

(f) A word u is in $(L \cup H) \cap K$ iff (u is in L or u is in H) and u is in K iff (u is in L and u is in K) or (u in in H and u is in K) iff $u \in (L \cap K) \cup (H \cap K)$. Thus the first law of distributivity follows from the distributivity of "and" and "or" in the logical setting. The second law of distributivity given as $(L \cap H) \cup K = (L \cup K) \cap (H \cup K)$ is proven similarly.

(g) The equation $(L \cup H) - K = (L - K) \cup (H - K)$ is equal to $(L \cup H) \cap (\Sigma^* - K) = (L \cap (\Sigma^* - K)) \cup (H \cap (\Sigma^* - K))$ and can be mapped back to **(f)** by using $\Sigma^* - K$ in place of K where Σ is the base alphabet of the languages considered. Furthermore, a word u is in $(L \cap H) - K$ iff u is in both L and H but not in K iff u is in both $L - K$ and $H - K$ iff u is in $(L - K) \cap (H - K)$. ∎

Proposition 1.16. *The following inequality rules apply to any sets and the mentioned inclusions / inequalities are proper for the examples provided:*

(a) $L \cdot L$ *can be different from* L: $\{0\} \cdot \{0\} = \{00\}$;

(b) $(L \cap H)^* \subseteq L^* \cap H^*$;
Properness: $L = \{00\}$, $H = \{000\}$, $(L \cap H)^* = \{\varepsilon\}$, $L^* \cap H^* = \{000000\}^*$;

(c) *If* $\{\varepsilon\} \cup (L \cdot H) = H$ *then* $L^* \subseteq H$;
Properness: $L = \{\varepsilon\}$, $H = \{0\}^*$;

(d) *If* $L \cup (L \cdot H) = H$ *then* $L^+ \subseteq H$;
Properness: $L = \{\varepsilon\}$, $H = \{0\}^*$;

(e) $(L \cap H) \cdot K \subseteq (L \cdot K) \cap (H \cdot K)$;
Properness: $(\{0\} \cap \{00\}) \cdot \{0, 00\} = \emptyset \subseteq \{000\} = (\{0\} \cdot \{0, 00\}) \cap (\{00\} \cdot \{0, 00\})$;

(f) $K \cdot (L \cap H) \subseteq (K \cdot L) \cap (K \cdot H)$;
Properness: $\{0, 00\} \cdot (\{0\} \cap \{00\}) = \emptyset \subseteq \{000\} = (\{0, 00\} \cdot \{0\}) \cap (\{0, 00\} \cdot \{00\})$.

Proof. Item (a) and the witnesses for the properness of the inclusions in items (b)–(f).

For the inclusion in (b), assume that $v = w_1 w_2 \ldots w_n$ is in $(L \cap H)^*$ with $w_1, w_2, \ldots, w_n \in (L \cap H)$; $v = \varepsilon$ in the case that $n = 0$. Now $w_1, w_2, \ldots, w_n \in L$ and therefore $v \in L^*$; $w_1, w_2, \ldots, w_n \in H$ and therefore $v \in H^*$; thus $v \in L^* \cap H^*$.

For items (c) and (d), define inductively $L^0 = \{\varepsilon\}$ and $L^{n+1} = L^n \cdot L$; equivalently one could say $L^{n+1} = L \cdot L^n$. It follows from the definition that $L^* = \bigcup_{n \geq 0} L^n$ and $L^+ = \bigcup_{n \geq 1} L^n$. In item (c), $\varepsilon \in H$ and thus $L^0 \subseteq H$. Inductively, if $L^n \subseteq H$ then $L^{n+1} = L \cdot L^n \subseteq L \cdot H \subseteq H$; thus $\bigcup_{n \geq 0} L^n \subseteq H$, that is, $L^* \subseteq H$. In item (d), $L^1 \subseteq H$ by definition. Now, inductively, if $L^n \subseteq H$ then $L^{n+1} = L \cdot L^n \subseteq L \cdot H \subseteq H$. Now $L^+ = \bigcup_{n \geq 1} L^n \subseteq H$.

The proofs of items (e) and (f) are similar, so just the proof of (e) is given here. Assume that $u \in (L \cap H) \cdot K$. Now $u = vw$ for some $v \in L \cap H$ and $w \in K$. It follows that $v \in L$ and $v \in H$, thus $vw \in L \cdot K$ and $vw \in H \cdot K$. Thus $u = vw \in (L \cdot K) \cap (H \cdot K)$. ∎

The proofs of (c) and (d) actually show also the following: If $\{\varepsilon\} \cup (L \cdot H) \subseteq H$ then $L^* \subseteq H$; if $L \cup (L \cdot H) \subseteq H$ then $L^+ \subseteq H$. Furthermore, $H = L^*$ and $H = L^+$ satisfies $\{\varepsilon\} \cup (L \cdot H) = H$ $L \cup (L \cdot H) = H$, respectively. Thus one has the following corollary.

Corollary 1.17. *For any set L, the following statements characterise L^* and L^+:*

(a) *L^* is the smallest set H such that $\{\varepsilon\} \cup (L \cdot H) = H$;*
(b) *L^* is the smallest set H such that $\{\varepsilon\} \cup (L \cdot H) \subseteq H$;*
(c) *L^+ is the smallest set H such that $L \cup (L \cdot H) = H$;*
(d) *L^+ is the smallest set H such that $L \cup (L \cdot H) \subseteq H$.*

Exercise 1.18. *Which three of the following sets are not equal to any of the other sets:*

(a) $\{01, 10, 11\}^*$*;*
(b) $((\{0,1\} \cdot \{0,1\}) - \{00\})^*$*;*
(c) $(\{01, 10\} \cdot \{01, 10, 11\} \cup \{01, 10, 11\} \cdot \{01, 10\})^*$*;*
(d) $(\{01, 10, 11\} \cdot \{01, 10, 11\})^* \cup \{01, 10, 11\} \cdot (\{01, 10, 11\} \cdot \{01, 10, 11\})^*$*;*
(e) $\{0,1\}^* - \{0,1\} \cdot \{00, 11\}^*$*;*
(f) $((\{01\}^* \cup \{10\})^* \cup \{11\})^*$*;*
(g) $(\{\varepsilon\} \cup (\{0\} \cdot \{0,1\}^* \cap \{1\} \cdot \{0,1\}^*))^*$.

Explain the answer.

Exercise 1.19. *Make a regular expression which contains all those decimal natural numbers which start with* 3 *or* 8 *and have an even number of digits and end with* 5 *or* 7.

Make a further regular expression which contains all odd ternary numbers without leading 0*s; here a ternary number is a number using the digits* $0, 1, 2$ *with* 10 *being three,* 11 *being four and* 1212 *being fifty. The set described should contain the ternary numbers* $1, 10, 12, 21, 100, 102, 111, 120, 122, 201, \ldots$ *which are the numbers* $1, 3, 5, 7, 9, 11, 13, 15, 17, 19, \ldots$ *in decimal.*

Exercise 1.20. *Let S be the smallest class of languages such that*

- *every language of the form u^* for a non-empty word u is in S;*
- *the union of two languages in S is again in S;*
- *the concatenation of two languages in S is again in S.*

Prove by structural induction the following properties of S:

(a) *Every language in S is infinite;*
(b) *Every language in S has polynomial growth.*

Lay out all inductive steps explicitly without only citing results in this lecture.
The set of binary numbers (without leading zeroes) can be described by the regular expression $\{0\} \cup (\{1\} \cdot \{0,1\}^*)$. Alternatively, one could describe these numbers also in a recursive way as the following example shows.

Example 1.21. If one wants to write down a binary number, one has the following recursive rules:

- A binary number can just be the string "0";
- A binary number can be a string "1" followed by some digits;

- Some digits can either be "0" followed by some digits or "1" followed by some digits or just the empty string.

So the binary number 101 consists of a 1 followed by some digits. These some digits consists of a 0 followed by some digits; now these some digits can again be described as a 1 followed by some digits; the remaining some digits are now void, so one can describe them by the empty string and the process is completed. Formally, one can use S to describe binary numbers and T to describe some digits and put the rules into this form:

- $S \to 0$;
- $S \to 1T$;
- $T \to T0$, $T \to T1$, $T \to \varepsilon$.

Now the process of making 101 is obtained by applying the rules iteratively: $S \to 1T$ to S giving $1T$; now $T \to 0T$ to the T in $1T$ giving $10T$; now $T \to 1T$ to the T in $10T$ giving $101T$; now $T \to \varepsilon$ to the T in $101T$ giving 101. Such a process is described by a grammar.

Grammars have been formalised by linguists as well as by mathematicians. They trace in mathematics back to Thue [75] and in linguistics, Chomsky [15] was one of the founders. Thue mainly considered a set of strings over a finite alphabet Σ with rules of the form $l \to r$ such that every string of the form xly can be transformed into xry by applying that rule. A Thue-system is given by a finite alphabet Σ and a finite set of rules where for each rule $l \to r$ also the rule $r \to l$ exists; a semi-Thue-system does not need to permit for each rule also the inverted rule. Grammars are in principle semi-Thue-systems, but they have made the process of generating the words more formal. The main idea is that one has additional symbols, so called non-terminal symbols, which might occur in the process of generating a word but which are not permitted to be in the final word. In the introductory example, S (binary numbers) and T (some digits) are the non-terminal symbols and $0, 1$ are the terminal digits. The formal definition is the following.

Definition 1.22. *A grammar (N, Σ, P, S) consists of two disjoint finite sets of symbols N and Σ, a set of rules P and a starting symbol $S \in N$.*

Each rule is of the form $l \to r$ where l is a string containing at least one symbol from N.

v can be derived from w in one step iff there are x, y and a rule $l \to r$ such that $v = xly$ and $w = xrw$. v can be derived from w in arbitrary steps iff there are $n \geq 0$ and $u_0, u_1, \ldots, u_n \in (N \cup \Sigma)^*$ such that $u_0 = v$, $u_n = w$ and u_{m+1} can be derived from

u_m in one step for each $m < n$.

Now (N, Σ, P, S) generates the set $L = \{w \in \Sigma^* : w$ can be derived from $S\}$.

Convention. One writes $v \Rightarrow w$ for saying that w can be derived from v in one step and $v \Rightarrow^* w$ for saying that w can be derived from v (in an arbitrary number of steps).

Example 1.23. Let $N = \{S, T\}$, $\Sigma = \{0, 1\}$, P contain the rules $S \to 0T1, T \to 0T, T \to T1, T \to 0, T \to 1$ and S be the start symbol.

Then $S \Rightarrow^* 001$ and $S \Rightarrow^* 011$: $S \Rightarrow 0T1 \Rightarrow 001$ and $S \Rightarrow 0T1 \Rightarrow 011$ by applying the rule $S \to 0T1$ first and then either $T \to 0$ or $T \to 1$. Furthermore, $S \Rightarrow^* 0011$ by $S \Rightarrow 0T1 \Rightarrow 0T11 \Rightarrow 0011$, that is, by applying the rules $S \to 0T1$, $T \to T1$ and $T \to 0$. $S \not\Rightarrow^* 000$ and $S \not\Rightarrow^* 111$ as the first rule must be $S \to 0T1$ and any word generated will preserve the 0 at the beginning and the 1 at the end.

This grammar generates the language of all strings which have at least 3 symbols and which consist of 0s followed by 1s where there must be at least one 0 and one 1.

Example 1.24. Let $(\{S\}, \{0, 1\}, P, S)$ be a grammar where P consists of the four rules $S \to SS|0S1|1S0|\varepsilon$.

Then $S \Rightarrow^* 0011$ by applying the rule $S \to 0S1$ twice and then applying $S \to \varepsilon$. Furthermore, $S \Rightarrow^* 010011$ which can be seen as follows: $S \Rightarrow SS \Rightarrow 0S1S \Rightarrow 01S \Rightarrow 010S1 \Rightarrow 0100S11 \Rightarrow 010011$.

This grammar generates the language of all strings in $\{0, 1\}^*$ which contain as many 0s as 1s.

Example 1.25. Let $(\{S, T\}, \{0, 1, 2\}, P, S)$ be a grammar where P consists of the rules $S \to 0T|1T|2T|0|1|2$ and $T \to 0S|1S|2S$.

Then $S \Rightarrow^* w$ iff $w \in \{0, 1, 2\}^*$ and the length of w is odd; $T \Rightarrow^* w$ iff $w \in \{0, 1, 2\}^*$ and the length of w is even but not 0.

This grammar generates the language of all strings over $\{0, 1, 2\}$ which have an odd length.

Exercise 1.26. *Make a grammar which generates all strings with four* 1*s followed by one* 2 *and arbitrary many* 0*s in between. That is, the grammar should correspond to the regular expression* $0^*10^*10^*10^*10^*20^*$.

The Chomsky Hierarchy. Noam Chomsky [15] studied the various types of grammars and introduced the hierarchy named after him; other pioneers of the theory of formal languages include Marcel-Paul Schützenberger. The Chomsky hierarchy has four main levels; these levels were later refined by introducing and investigating other classes of grammars and formal languages defined by them.

Definition 1.27. *Let (N, Σ, P, S) be a grammar. The grammar belongs to the first of the following levels of the Chomsky hierarchy which applies:*

(CH3) *The grammar is called left-linear (or regular) if every rule (member of P) is of the form $A \to wB$ or $A \to w$ where A, B are non-terminals and $w \in \Sigma^*$. A language is regular iff it is generated by a regular grammar.*

(CH2) *The grammar is called context-free iff every rule is of the form $A \to w$ with $A \in N$ and $w \in (N \cup \Sigma)^*$. A language is context-free iff it is generated by a context-free grammar.*

(CH1) *The grammar is called context-sensitive iff every rule is of the form $uAw \to uvw$ with $A \in N$ and $u, v, w \in (N \cup \Sigma)^+$; furthermore, in the case that the start symbol S does not appear on any right side of a rule, the rule $S \to \varepsilon$ can be added so that the empty word can be generated. A language is called context-sensitive iff it is generated by a context-sensitive grammar.*

(CH0) *There is the most general case where the grammar does not satisfy any of the three restrictions above. A language is called recursively enumerable iff it is generated by some grammar.*

The next theorem permits easier methods to prove that a language is context-sensitive by constructing the corresponding grammars.

Theorem 1.28. *A language L not containing ε is context-sensitive iff it can be generated by a grammar (N, Σ, P, S) satisfying that every rule $l \to r$ satisfies $|l| \leq |r|$.*

A language L containing ε is context-sensitive iff it can be generated by a grammar (N, Σ, P, S) satisfying that $S \to \varepsilon$ is a rule and that any further rule $l \to r$ satisfies $|l| \leq |r| \wedge r \in (N \cup \Sigma - \{S\})^*$.

Example 1.29. The grammar $(\{S, T, U\}, \{0, 1, 2\}, P, S)$ with P consisting of the rules $S \to 0T12|012|\varepsilon$, $T \to 0T1U|01U$, $U1 \to 1U$, $U2 \to 22$ generates the language of all strings $0^n1^n2^n$ where n is a natural number (including 0).

For example, $S \Rightarrow 0T12 \Rightarrow 00T1U12 \Rightarrow 00T11U2 \Rightarrow 00T1122 \Rightarrow 0001U1122 \Rightarrow 00011U122 \Rightarrow 000111U22 \Rightarrow 000111222$.

One can also see that the numbers of the 0s, 1s and 2s generated are always the same: the rules $S \to 0T12$ and $S \to 012$ and $S \to \varepsilon$ produce the same quantity of these symbols; the rules $T \to 0T1U$ and $T \to 01U$ produce one 0, one 1 and one U which can only be converted into a 2 using the rule $U2 \to 22$ but cannot be converted into anything else; it must first move over all 1s using the rule $U1 \to 1U$ in order to meet a 2 which permits to apply $U2 \to 22$. Furthermore, one can see that the resulting string has always the 0s first, followed by 1s and the 2s last. Hence every string generated is of the form $0^n1^n2^n$.

Note that the notion of regular language is the same whether it is defined by a regular grammar or by a regular expression.

Theorem 1.30. *A language L is generated by a regular expression iff it is generated by a left-linear grammar.*

Proof. One shows by induction that every language generated by a regular expression is also generated by a regular grammar. A finite language $\{w_1, w_2, \ldots, w_n\}$ is generated by the grammar with the rules $S \to w_1|w_2|\ldots|w_n$. For the inductive sets, assume now that L and H are regular sets (given by regular expressions) which are generated by the grammars (N_1, Σ, P_1, S_1) and (N_2, Σ, P_2, S_2), where the sets of non-terminals are disjoint: $N_1 \cap N_2 = \emptyset$. Now one can make a grammar $(N_1 \cup N_2 \cup \{S, T\}, \Sigma, P, S)$ where P depends on the respective case of $L \cup H$, $L \cdot H$ and L^*. The set P of rules (with A, B being non-terminals and w being a word of terminals) is defined as follows in the respective case:

Union $L \cup H$: P contains all rules from $P_1 \cup P_2$ plus $S \to S_1|S_2$;

Concatenation $L \cdot H$: P contains the rules $S \to S_1$, $T \to S_2$ plus all rules of the form $A \to wB$ which are in $P_1 \cup P_2$ plus all rules of the form $A \to wT$ with $A \to w$ in P_1 plus all rules of the form $A \to w$ in P_2;

Kleene Star L^*: P contains the rules $S \to S_1$ and $S \to \varepsilon$ and each rule $A \to wB$ which is in P_1 and each rule $A \to wS$ for which $A \to w$ is in P_1.

It is easy to see that in the case of the union, a word w can be generated iff one uses the rule $S \to S_1$ and $S_1 \Rightarrow^* w$ or one uses the rule $S \to S_2$ and $S_2 \Rightarrow^* w$. Thus $S \Rightarrow^* w$ iff $w \in L$ or $w \in H$.

In the case of a concatenation, a word u can be generated iff there are v, w such that $S \Rightarrow^* S_1 \Rightarrow^* vT \Rightarrow vS_2 \Rightarrow^* vw$ and $u = vw$. This is the case iff L contains v and H contains w: $S_1 \Rightarrow^* vT$ iff one can, by same rules with only the last one changed to have the final T omitted derive that $v \in L$ for the corresponding grammar; $T \Rightarrow^* w$ iff one can derive in the grammar for H that $w \in L$. Here T was introduced for being able to give this formula; one cannot use S_2 directly as the grammar for H might permit that $S_2 \Rightarrow^* tS_2$ for some non-empty word t.

The ingredient for the verification of the grammar for Kleene star is that $S_1 \to uS$ without using the rule $S \to S_1$ iff $S_1 \to u$ can be derived in the original grammar for L; now one sees that $S \to^* uS$ for non-empty words in the new grammar is only possible iff $u = u_1 u_2 \ldots u_n$ for some n and words $u_1, u_2, \ldots, u_n \in L$; furthermore, the empty word can be generated.

For the converse direction, assume that a regular grammar with rules $R_1, R_2, \ldots, R_n$ is given. One makes a sequence of regular expressions $E_{C,D,m}$ and $E_{C,m}$ where C, D are any non-terminals and which will satisfy the following conditions:

- $E_{C,D,m}$ generates the language of words v for which there is a derivation $C \Rightarrow^* vD$ using only the rules $R_1, R_2, \ldots, R_m$;
- $E_{C,m}$ generates the language of all words v for which there is a derivation $C \Rightarrow^* v$ using only the rules $R_1, R_2, \ldots, R_m$.

One initialises all $E_{C,0} = \emptyset$ and if $C = D$ then $E_{C,D} = \{\varepsilon\}$ else $E_{C,D} = \emptyset$. If $E_{C,m}$ and $E_{C,D,m}$ are defined for $m < n$, then one defines the expressions $E_{C,m+1}$ and $E_{C,D,m+1}$ in dependence of what R_{m+1} is.

If R_{m+1} is of the form $A \to w$ for a non-terminal A and a terminal word w then one defines the updated sets as follows for all C, D:

- $E_{C,D,m+1} = E_{C,D,m}$, as one cannot derive anything ending with D with help of R_{m+1} what can not already be derived without help of R_{m+1};
- $E_{C,m+1} = E_{C,m} \cup (E_{C,A,m} \cdot \{w\})$, as one can either only use old rules what is captured by $E_{C,m}$ or go from C to A using the old rules and then terminating the derivation with the rule $A \to w$.

In both cases, the new expression is used by employing unions and concatenations and thus is in both cases again a regular expression.

If R_{m+1} is of the form $A \to wB$ for non-terminals A, B and a terminal word w then one defines the updated sets as follows for all C, D:

- $E_{C,D,m+1} = E_{C,D,m} \cup E_{C,A,m} \cdot w \cdot (E_{B,A,m} \cdot w)^* \cdot E_{B,D,m}$, as one can either directly go from C to D using the old rules or go to A employing the rule and producing a w and then ending up in B with a possible repetition by going be to A and employing again the rule making a w finitely often and then go from B to D;
- $E_{C,m+1} = E_{C,m} \cup E_{C,A,m} \cdot w \cdot (E_{B,A,m} \cdot w)^* \cdot E_{B,m}$, as one can either directly generate a terminal word using the old rules or go to A employing the rule and producing a w and then ending up in B with a possible repetition by going be to A and employing again the rule making a w finitely often and then employ more rules to finalise the making of the word.

Again, the new regular expressions put together the old ones using union, concatenation and Kleene star only. Thus one obtains also on level $m+1$ a set of regular expressions.

After one has done this by induction for all the rules in the grammar, the resulting expression $E_{S,n}$ where S is the start symbol generates the same language as the given grammar did. This completes the second part of the proof. ∎

For small examples, one can write down the languages in a more direct manner, though it is still systematic.

Example 1.31. Let L be the language $(\{0,1\}^* \cdot 2 \cdot \{0,1\}^* \cdot 2) \cup \{0,2\}^* \cup \{1,2\}^*$.

A regular grammar generating this language is $(\{S,T,U,V,W\},\{0,1,2\},P,S)$ with the rules $S \to T|V|W$, $T \to 0T|1T|2U$, $U \to 0U|1U|2$, $V \to 0V|2V|\varepsilon$ and $W \to 1W|2W|\varepsilon$.

Using the terminology of Example 1.33, $L_U = \{0,1\}^* \cdot 2$, $L_T = \{0,1\}^* \cdot 2 \cdot L_U = \{0,1\}^* \cdot 2 \cdot \{0,1\}^* \cdot 2$, $L_V = \{0,2\}^*$, $L_W = \{1,2\}^*$ and $L = L_S = L_T \cup L_V \cup L_W$.

Exercise 1.32. *Let L be the language $(\{00,11,22\} \cdot \{33\}^*)^*$. Make a regular grammar generating the language.*

Example 1.33. Let $(\{S,T\},\{0,1,2,3\},P,S)$ be a given regular grammar.

For $A,B \in \{S,T\}$, let $L_{A,B}$ be the finite set of all words $w \in \{0,1,2,3\}^*$ such that the rule $A \to wB$ exists in P and let L_A be the finite set of all words $w \in \{0,1,2,3\}^*$ such that the rule $A \to w$ exists in P. Now the grammar generates the language

$$(L_{S,S})^* \cdot (L_{S,T} \cdot (L_{T,T})^* \cdot L_{T,S} \cdot (L_{S,S})^*)^* \cdot (L_S \cup L_{S,T} \cdot (L_{T,T})^* \cdot L_T).$$

For example, if P contains the rules $S \to 0S|1T|2$ and $T \to 0T|1S|3$ then the language generated is

$$0^* \cdot (10^*10^*)^* \cdot (2 \cup 10^*3)$$

which consists of all words from $\{0,1\}^* \cdot \{2,3\}$ such that either the number of 1s is even and the word ends with 2 or the number of 1s is odd and the word ends with 3.

Exercise 1.34. *Let $(\{S,T,U\},\{0,1,2,3,4\},P,S)$ be a grammar where the set P contains the rules $S \to 0S|1T|2$, $T \to 0T|1U|3$ and $U \to 0U|1S|4$. Make a regular expression describing this language.*

The Pumping Lemmas are methods to show that certain languages are not regular or not context-free. These criteria are only sufficient to show that a language is more complicated than assumed, they are not necessary. The following version is the standard version of the pumping lemma.

Theorem 1.35: Pumping Lemma. **(a)** *Let $L \subseteq \Sigma^*$ be an infinite regular language. Then there is a constant k such that for every $u \in L$ of length at least k there is a representation $x \cdot y \cdot z = u$ such that $|xy| \leq k$, $y \neq \varepsilon$ and $xy^*z \subseteq L$.*

(b) *Let $L \subseteq \Sigma^*$ be an infinite context-free language. Then there is a constant k such that for every $u \in L$ of length at least k there is a representation $vwxyz = u$ such that $|wxy| \leq k$, $w \neq \varepsilon \vee y \neq \varepsilon$ and $vw^\ell xy^\ell z \in L$ for all $\ell \in \mathbb{N}$.*

Proof. Part (a): One considers for this proof only regular expressions might up by finite sets and unions, concatenations and Kleene star of other expressions. For regular expressions σ, let $L(\sigma)$ be the language described by σ. Now assume that σ is a shortest regular expression such that for $L(\sigma)$ fails to satisfy the Pumping Lemma. One of the following cases must apply to σ:

First, $L(\sigma)$ is a finite set given by an explicit list in σ. Let k be a constant longer than every word in $L(\sigma)$. Then the Pumping Lemma would be satisfied as it only requests any condition on words in L which are longer than k – there are no such words.

Second, σ is $(\tau \cup \rho)$ for further regular expressions τ, ρ. As τ, ρ are shorter than σ, $L(\tau)$ satisfies the Pumping Lemma with constant k' and $L(\rho)$ with constant k''; let $k = \max\{k', k''\}$. Consider any word $w \in L(\sigma)$ which is longer than k. If $w \in L(\tau)$ then $|w| > k'$ and $w = xyz$ for some x, y, z with $y \neq \varepsilon$ and $|xy| \leq k'$ and $xy^*z \subseteq L(\tau)$. It follows that $|xy| \leq k$ and $xy^*z \subseteq L(\sigma)$. Similarly, if $w \in L(\rho)$ then $|w| > k''$ and $w = xyz$ for some x, y, z with $y \neq \varepsilon$ and $|xy| \leq k''$ and $xy^*z \subseteq L(\rho)$. It again follows that $|xy| \leq k$ and $xy^*z \subseteq L(\sigma)$. Thus the Pumping Lemma also holds in this case with the constant $k = \max\{k', k''\}$.

Third, σ is $(\tau \cdot \rho)$ for further regular expressions τ, ρ. As τ, ρ are shorter than σ, $L(\tau)$ satisfies the Pumping Lemma with constant k' and $L(\rho)$ with constant k''; let $k = k' + k''$. Consider any word $u \in L(\sigma)$ which is longer than k. Now $u = vw$ with $v \in L(\tau)$ and $w \in L(\rho)$. If $|v| > k'$ then $v = xyz$ with $y \neq \varepsilon$ and $|xy| \leq k'$ and $xy^*z \subseteq L(\tau)$. It follows that $|xy| \leq k$ and $xy^*(zw) \subseteq L(\sigma)$, so the Pumping Lemma is satisfied with constant k in the case $|v| > k'$. If $|v| \leq k'$ then $w = xyz$ with $y \neq \varepsilon$ and $|xy| \leq k''$ and $xy^*z \subseteq L(\rho)$. It follows that $|(vx)y| \leq k$ and $(vx)y^*z \subseteq L(\sigma)$, so the Pumping Lemma is satisfied with constant k in the case $|v| \leq k'$ as well.

Fourth, σ is τ^* for further regular expression τ. Then τ is shorter than σ and $L(\tau)$ satisfies the Pumping Lemma with some constant k. Now it is shown that $L(\sigma)$ satisfies the Pumping Lemma with the same constant k. Assume that $v \in L(\sigma)$ and $|v| > k$. Then $v = w_1 w_2 \ldots w_n$ for some $n \geq 1$ and non-empty words $w_1, w_2, \ldots, w_n \in L(\tau)$. If $|w_1| \leq k$ then let $x = \varepsilon$, $y = w_1$ and $z = w_2 \cdot \ldots \cdot w_n$. Now $xy^*z = w_1^* w_2 \ldots w_n \subseteq L(\tau)^* = L(\sigma)$. If $|w_1| > k$ then there are x, y, z with $w_1 = xyz$, $|xy| \leq k$, $y \neq \varepsilon$ and $xy^*z \subseteq L(\tau)$. It follows that $xy^*(z \cdot w_2 \cdot \ldots \cdot w_n) \subseteq L(\sigma)$. Again the Pumping Lemma is satisfied.

It follows from this case distinction that the Pumping Lemma is satisfied in all cases and therefore the regular expression σ cannot be exist as assumed. Thus all regular languages satisfy the Pumping Lemma.

Part (b) is omitted; see the lecture notes on Theory of Computation. ∎

In Section 2 below a more powerful version of the pumping lemma for regular sets will be shown. The following weaker corollary might also be sufficient in some cases to show that a language is not regular.

Corollary 1.36. *Assume that L is an infinite regular language. Then there is a constant k such that for each word $w \in L$ with $|w| > k$, one can represent w as $xyz = w$ with $y \neq \varepsilon$ and $xy^*z \subseteq L$.*

Exercise 1.37. *Let $p_1, p_2, p_3, \ldots$ be the list of prime numbers in ascending order. Show that $L = \{0^n : n > 0$ and $n \neq p_1 \cdot p_2 \cdot \ldots \cdot p_m$ for all $m\}$ satisfies Corollary 1.36 but does not satisfy Theorem 1.35 (a).*

Exercise 1.38. *Assume that (N, Σ, P, S) is a regular grammar and h is a constant such that N has less than h elements and for all rules of the form $A \to wB$ or $A \to w$ with $A, B \in N$ and $w \in \Sigma^*$ it holds that $|w| < h$. Show that Theorem 1.35 (a) holds with the constant k being h^2.*

Exercise 1.39. *Prove a weaker version of Theorem 1.35 (b) without requesting that the $|wxy| \leq k$. The idea to be used is that there is a constant k such that for every word $u \in L$ which is longer than k, one can be split into $vwxyz$ such that there is a non-terminal A with $S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vwxyz$. Then $A \Rightarrow^* wAy$ is equivalent to $vAz \Rightarrow^* vwAyz$ and use this fact to derive the pumping lemma.*

Example 1.40. The set $L = \{0^p : p$ is a prime number$\}$ of all 0-strings of prime length is not context-free.

To see this, assume the contrary and assume that k is the constant from the pumping condition in Theorem 1.35 (b). Let p be a prime number larger than k. Then 0^p can be written in the form $vwxyz$ with $q = |wy| > 0$. Then every string of the form $vw^\ell xy^\ell z$ is in L; these strings are of the form $0^{p+q\cdot(\ell-1)}$. Now choose $\ell = p+1$ and consider $0^{p+q\cdot p}$. The number $p + q \cdot p = p \cdot (q+1)$ is not a prime number; however $0^{p+q\cdot p}$ is in L by the pumping condition in Theorem 1.35 (b). This contradiction proves that L cannot be context-free.

Example 1.41. The language L of all words which have as many 0 as 1 satisfies the pumping condition in Corollary 1.36 but not the pumping condition in Theorem 1.35 (a).

For seeing the first, note that whenever w has as many 0 as 1 then every element of w^* has the same property. Indeed, $L = L^*$ and Corollary 1.36 is satisfied by every language which is of the form H^* for some H.

For seeing the second, assume the contrary and assume that n is the constant used in Theorem 1.35 (a). Now consider the word 0^n1^n. By assumption there is a representation $xyz = 0^n1^n$ with $|xy| \leq n$ and $y \neq \varepsilon$. As a consequence, $xyyz = 0^{n+m}1^n$ for some $m > 0$ and $xyyz \notin L$. Hence the statement in Theorem 1.35 (a) is not satisfied.

Theorem 1.42. *Let $L \subseteq \{0\}^*$. The following conditions are equivalent for L:*

(a) *L is regular;*
(b) *L is context-free;*
(c) *L satisfies the Theorem 1.35 (a) for regular languages;*
(d) *L satisfies the Theorem 1.35 (b) for context-free languages.*

Proof. Clearly (a) implies (b),(c) and (b),(c) both imply (d). Now it will be shown that (d) implies (a).

Assume that k is the pumping constant for the context-free Pumping Lemma. Then, for every word $u \in L$, one can split 0^n into $vwxyz$ such that $|wxy| \leq k$ and at least one of w, y is not empty and $vw^hxy^hz \in L$ for all h.

Now when h is a multiple of $k!/|wy|$, the word vw^hxy^hz is equal to $0^n \cdot 0^{k! \cdot (h \cdot |wy|/k!)}$ where $h \cdot |wy|/k!$ can be any integer, depending on the choice of h. As all these vw^hxy^hz are in L, it follows that $0^n \cdot (0^{k!})^* \subseteq L$. For each remainder $m \in \{0, 1, \ldots, k! - 1\}$, let

$$n_m = \min\{i : \exists j\, [i > k \textit{ and } i = m + jk! \textit{ and } 0^i \in L]\}$$

and let $n_m = \infty$ when there is no such i, that is, $\min \emptyset = \infty$.

Now L is the union of finitely many regular sets: First the set $L \cap \{\varepsilon, 0, 00, \ldots, 0^k\}$ which is finite and thus regular. Second, all those sets $0^{n_m} \cdot (0^{k!})^*$ where $m < k!$ and $n_m < \infty$. There are at most $k!$ many of these sets of the second type and each is given by a regular expression. Thus L is the union of finitely many regular sets and therefore regular itself. ∎

Exercise 1.43. *Consider the following languages:*

- $L = \{0^n1^n2^n : n \in \mathbb{N}\}$;
- $H = \{0^n1^m : n^2 \leq m \leq 2n^2\}$;

- $K = \{0^n 1^m 2^k : n \cdot m = k\}$.

Show that these languages are not context-free using Theorem 1.35 (b).

Exercise 1.44. *Construct a context-sensitive grammar for* $\{10^n 1 : n$ *is a power of three*$\}$. *Here the powers of three are* $1, 3, 9, 27, \ldots$ *and include the zeroth power of three.*

Exercise 1.45. *Construct a context-sensitive grammar for* $\{10^n 1 : n$ *is at least four and not a prime*$\}$.

Exercise 1.46. *Construct a context-free grammar for the language* $\{uvw \in \{0,1\}^* : |u| = |v| = |w|$ *and* $(u \neq v$ *or* $u \neq w$ *or* $v \neq w)\}$.

Exercise 1.47. *Let* $F(L) = \{v : \exists w \in L\,[v$ *is obtained by reordering the symbols in* $w]\}$. *Reorderings include the void reordering where all digits remain at their position. So* $F(\{0, 00, 01\}) = \{0, 00, 01, 10\}$. *Determine the possible levels in Chomsky hierarchy which* $F(L)$ *can have when* L *is regular. For each possible level, exhibit a regular language* L *such that* $F(L)$ *is exactly on that level.*

Exercise 1.48. *Let* $F(L)$ *as in Exercise 1.47 and consider the following weaker version of Theorem 1.35 (b): There is a constant* c *such that all words in* $u \in F(L)$ *with* $|u| \geq c$ *can be represented as* $u = v \cdot w \cdot x \cdot y \cdot z$ *with* $w \cdot y \neq \varepsilon$ *and* $vw^n xy^n z \in F(L)$ *for all* $n \in \mathbb{N}$. *Provide a language* L *such that* $F(L)$ *satisfies this weaker version of the pumping lemma but not Theorem 1.35 (b).*

Exercise 1.49. *Let* $L = \{0^n 1^m 2^k : n \cdot m = 9 \cdot k\}$. *Show that* L *satisfies the pumping lemma from Exercise 1.48 while* $F(L)$ *does not satisfy this pumping lemma where* F *is as in Exercise 1.47.*

Exercise 1.50. *For given* L, *let* $G(L) = \{vw : wv \in L$ *and* $v, w \in \Sigma^*\}$ *and note that* $L \subseteq G(L)$, *as it can be that* v *or* w *is* ε *in the above formula for* $G(L)$. *Provide all levels of the Chomsky hierarchy for which there is an* L *exactly on this level such that* $G(L)$ *is regular; note that when the membership problem of a language* L *cannot be solved by an algorithm in exponential time then* L *is not context-sensitive.*

Exercise 1.51. *Let* $L = \{w \in \{0,1,2,3\}^* :$ *if* $a < b$ *then* b *occurs more frequently than* $a\}$. *What is the exact level of* L *in the Chomsky hierarchy? Use grammars and pumping lemmas to prove the result.*

Exercise 1.52. *Let* L *be given by the grammar* $(\{S\}, \{0,1\}, \{S \to 01S|01, S0 \to 0S, S1 \to 1S, 0S \to S0, 1S \to S1\}, S)$. *Determine the level of* L *in the Chomsky* *hierarchy, it is one of regular, context-free and context-sensitive, as it is given by a context-sensitive grammar. Determine all words up to length* 6 *in* L *and explain verbally when a word belongs to* L.

Exercise 1.53. *Construct context-free grammars for the sets* $L = \{0^n 1^m 2^k : n < m \vee m < k\}$, $H = \{0^n 1^m 2^{n+m} : n, m \in \mathbb{N}\}$ *and* $K = \{w \in \{0,1,2\}^* : w$ *has a subword of the form* $20^n 1^n 2$ *for some* $n > 0$ *or* $w = \varepsilon\}$.

Which of the versions of the Pumping Lemma (Theorems 1.35 (a) and 1.35 (b) and Corollary 1.36) are satisfied by L, H *and* K, *respectively.*

Exercise 1.54. *Let* $L = \{0^h 1^i 2^j 3^k : (h \neq i$ *and* $j \neq k)$ *or* $(h \neq k$ *and* $i \neq j)\}$ *be given. Construct a context-free grammar for* L *and determine which of versions of the Pumping Lemma (Theorems 1.35 (a) and 1.35 (b) and Corollary 1.36) are satisfied by* L.

Exercise 1.55. *Consider the grammar* $(\{S\}, \{0,1,2,3\}, \{S \rightarrow 00S|S1|S2|3\}, S)$ *and construct for the language* L *generated by the grammar the following: a regular grammar for* L *and a regular expression for* L.

2 Finite Automata

An automaton is in general a mechanism which checks whether a word is in a given language. An automaton has a number of states which memorise some information. Here an example.

Example 2.1: Divisibility by 3. Let $a_0 a_1 \ldots a_n$ be a decimal number. One can check whether $a_0 a_1 \ldots a_n$ is a multiple of 3 by the following algorithm using a memory $s \in \{0, 1, 2\}$ and processing in step m the digit a_m. The memory s is updated accordingly.

Case s=0 : If $a_m \in \{0, 3, 6, 9\}$ then update $s = 0$;
if $a_m \in \{1, 4, 7\}$ then update $s = 1$;
if $a_m \in \{2, 5, 8\}$ then update $s = 2$.

Case s=1 : If $a_m \in \{0, 3, 6, 9\}$ then update $s = 1$;
if $a_m \in \{1, 4, 7\}$ then update $s = 2$;
if $a_m \in \{2, 5, 8\}$ then update $s = 0$.

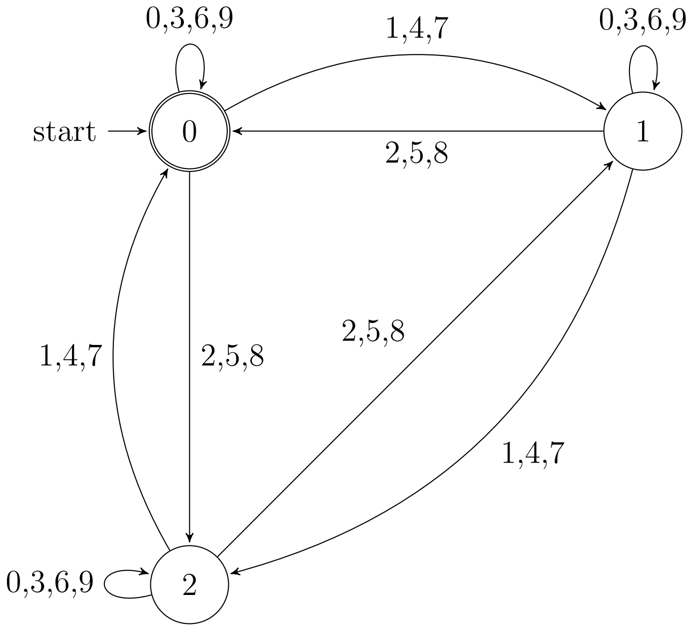
Case s=2 : If $a_m \in \{0, 3, 6, 9\}$ then update $s = 2$;
if $a_m \in \{1, 4, 7\}$ then update $s = 0$;
if $a_m \in \{2, 5, 8\}$ then update $s = 1$.

The number $a_0 a_1 \ldots a_n$ is divisible by 3 iff $s = 0$ after processing a_n. For example, 123456 is divisible by 3 as the value of s from the start up to processing the corresponding digits is $0, 1, 0, 0, 1, 0, 0$, respectively. The number 256 is not divisible by 3 and the value of s is $0, 2, 1, 1$ after processing the corresponding digits.

Quiz 2.2. *Which of the following numbers are divisible by* 3*:* 1*,* 20*,* 304*,* 2913*,* 49121*,* 391213*,* 2342342*,* 123454321*?*

Description 2.3: Deterministic Finite Automaton. The idea of this algorithm is to update a memory which takes only finitely many values in each step according to the digit read. At the end, it only depends on the memory whether the number which has been processed is a multiple of 3 or not. This is a quite general algorithmic method and it has been formalised in the notion of a finite automaton; for this, the possible values of the memory are called states. The starting state is the initial value of the memory. Furthermore, after processing the word it depends on the memory whether the word is in L or not; those values of the memory which say $a_0 a_1 \ldots a_n \in L$ are called "accepting states" and the others are called "rejecting states".

One can display the automata as a graph. The nodes of the graph are the states (possible values of the memory). The accepting states are marked with a double border, the rejecting states with a normal border. The indicator "start" or an incoming arrow mark the initial state. Arrows are labelled with those symbols on which a transition from one state to anothers takes place. Here the graphical representation of the automaton checking whether a number is divisible by 3.



Mathematically, one can also describe a finite automaton $(Q, \Sigma, \delta, s, F)$ as follows: Q is the set of states, Σ is the alphabet used, δ is the transition function mapping pairs from $Q \times \Sigma$ to Σ, s is the starting state and F is the set of accepting states.

The transition-function $\delta : Q \times \Sigma \to Q$ defines a unique extension with domain $Q \times \Sigma^*$ as follows: $\delta(q, \varepsilon) = q$ for all $q \in Q$ and, inductively, $\delta(q, wa) = \delta(\delta(q, w), a)$ for all $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$.

For any string $w \in \Sigma^*$, if $\delta(s, w) \in F$ then the automaton accepts w else the automaton rejects w.

Example 2.4. One can also describe an automaton by a table mainly maps down δ and furthermore says which states are accepting or rejecting. The first state listed is usually the starting state. Here a table for an automaton which checks whether a number is a multiple of 7:

q	type	$\delta(q, a)$ for $a = 0$	1	2	3	4	5	6	7	8	9
0	acc	0	1	2	3	4	5	6	0	1	2
1	rej	3	4	5	6	0	1	2	3	4	5
2	rej	6	0	1	2	3	4	5	6	0	1
3	rej	2	3	4	5	6	0	1	2	3	4
4	rej	5	6	0	1	2	3	4	5	6	0
5	rej	1	2	3	4	5	6	0	1	2	3
6	rej	4	5	6	0	1	2	3	4	5	6

This automaton checks whether a number is a multiple of 7.

On input 343 the automaton goes on symbol 3 from state 0 to state 3, then on symbol 4 from state 3 to state 2 and then on symbol 3 from state 6 to state 0. The state 0 is accepting and hence 343 is a multiple of 7 (in fact $343 = 7 * 7 * 7$).

On input 999 the state goes first from state 0 to state 2, then from state 2 to state 1, then from state 1 to state 5. The state 5 is rejecting and therefore 999 is not a multiple of 7 (in fact $999 = 7 * 142 + 5$).

Example 2.5. One can also describe a finite automaton as an update function which maps finite states plus symbols to finite states by some algorithm written in a more compact form. In general the algorithm has variables taking its values from finitely many possibilities and it can read symbols until the input is exhausted. It does not have arrays or variables which go beyond its finite range. It has explicit commands to accept or reject the input. When it does "accept" or "reject" the program terminates.

```
function div257
  begin var a in {0,1,2,...,256};
        var b in {0,1,2,3,4,5,6,7,8,9};
        if exhausted(input) then reject;
        read(b,input); a = b;
        if b == 0 then
          begin if exhausted(input) then accept else reject end;
        while not exhausted(input) do
          begin read(b,input); a = (a*10+b) mod 257 end;
        if a == 0 then accept else reject end.
```

This automaton checks whether a number on the input is a multiple of 257; furthermore, it does not accept any input having leading 0s. Here some sample runs of the algorithm.

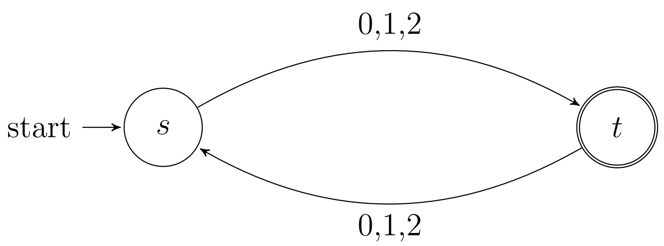
On input ε the algorithm rejects after the first test whether the input is exhausted. On input 00 the algorithm would read b one time and then do the line after the test whether b is 0; as the input is not yet exhausted, the algorithm rejects. On input 0 the algorithm goes the same way until but finally accepts the input as the input is exhausted after the symbol b has been read for the first time. On input 51657, the algorithm initialises a as 5 after having read b for the first time. Then it reaches the while-loop and, while reading $b = 1$, $b = 6$, $b = 5$, $b = 7$ it updates a to 51, 2, 25, 0, respectively. It accepts as the final value of a is 0. Note that the input 51657 is $201 * 257$ and therefore the algorithm is correct in this case.

Such algorithms permit to write automata with a large number of states in a more compact way then making a state diagram or a state table with hundreds of states.

Note that the number of states of the program is actually larger than 257, as not only the value of a but also the position in the program contributes to the state of the automaton represented by the program. The check "exhausted(input)" is there to check whether there are more symbols on the input to be processed or not; so the first check whether the input is exhausted is there to reject in the case that the input is the empty string. It is assumed that the input is always a string from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$.

Exercise 2.6. *Such an algorithm might be written in a form nearer to a finite automaton if one gives the set of states explicitly, names the starting state and the accepting states and then only places an algorithm or mathematical description in order to describe δ (in place of a table). Implement the above function div257 using the state space $Q = \{s, z, r, q_0, q_1, \ldots, q_{256}\}$ where s is the starting state and z, q_0 are the accepting states; all other states are rejecting. Write down how the transition-function δ is defined as a function from $Q \times \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \to Q$. Give a compact definition and not a graph or table.*

Quiz 2.7. *Let $(\{s, t\}, \{0, 1, 2\}, \delta, s, \{t\})$ be a finite automaton with $\delta(s, a) = t$ and $\delta(t, a) = s$ for all $a \in \{0, 1, 2\}$. Determine the language of strings recognised by this automaton.*



Theorem 2.8: Characterising Regular Sets. *If a language L is recognised by a deterministic finite automaton then L is regular.*

Proof. Let an automaton $(Q, \Sigma, \delta, s, F)$ be given. Now one builds the left-linear grammar (Q, Σ, P, s) with the following rules:

- the rule $q \to ar$ is in P iff $\delta(q, a) = r$;
- the rule $q \to \varepsilon$ is in P iff $q \in F$.

So the non-terminals of the grammar are the states of the automaton and also the roles of every $q \in Q$ is in both constructs similar: For all $q, r \in Q$, it holds that $q \Rightarrow^* wr$ iff $\delta(q, w) = r$.

To see this, one proves it by induction. First consider $w = \varepsilon$. Now $q \Rightarrow^* wr$ iff $q = r$ iff $\delta(q, w) = r$. Then consider $w = va$ for some symbol a and assume that the statement is already proven for the shorter word v. Now $q \Rightarrow^* wr$ iff there is a non-terminal t with $q \Rightarrow^* vt \Rightarrow var$ iff there is a non-terminal t with $\delta(q, v) = t$ and $t \Rightarrow ar$ iff there is a non-terminal t with $\delta(q, v) = t$ and $\delta(t, a) = r$ iff $\delta(q, w) = r$.

The only way to produce a word w in the new grammar is to generate the word wq for some $q \in F$ and then to apply the rule $q \to \varepsilon$. Thus, the automaton accepts w iff $\delta(s, w) \in F$ iff there is a $q \in F$ with $s \Rightarrow^* q \wedge q \Rightarrow \varepsilon$ iff $s \Rightarrow^* w$. Hence w is accepted by the automaton iff w is generated by the corresponding grammar. ∎

The converse of this theorem will be shown later in Theorem 2.47.

There is a stronger version of the pumping lemma which directly comes out of the characterisation of regular languages by automata; it is called the "Block Pumping Lemma", as it says that when a word in a regular language is split into sufficiently many blocks then one can pump one non-empty sequence of these blocks.

Theorem 2.9: Block Pumping Lemma. *If L is a regular set then there is a constant k such that for all strings $u_0, u_1, \ldots, u_k$ with $u_0 u_1 \ldots u_k \in L$ and $u_1, \ldots, u_{k-1}$ being nonempty there are i, j with $0 < i < j \leq k$ and*

$$(u_0 u_1 \ldots u_{i-1}) \cdot (u_i u_{i+1} \ldots u_{j-1})^* \cdot (u_j u_{j+1} \ldots u_k) \subseteq L.$$

So if one splits a word in L into $k + 1$ parts then one can select some parts in the middle of the word which can be pumped.

Proof. Given a regular set L, let $(Q, \Sigma, \delta, s, F)$ be the finite automaton recognising this language. Let $k = |Q| + 1$ and consider any strings $u_0, u_1, \ldots, u_k$ with $u_0 u_1 \ldots u_k \in L$. There are i and j with $0 < i < j \leq k$ such that $\delta(s, u_0 u_1 \ldots u_{i-1}) = \delta(s, u_0 u_1 \ldots u_{j-1})$; this is due to the fact that there are $|Q| + 1$ many values for i, j and so two of the states have to be equal. Let $q = \delta(s, u_0 u_1 \ldots u_{i-1})$. By assumption, $q = \delta(q, u_i u_{i+1} \ldots u_{j-1})$ and so it follows that $q = \delta(s, u_0 u_1 \ldots u_{i-1} (u_i u_{i+1} \ldots u_{j-1})^h)$ for every h. Furthermore, $\delta(q, u_j u_{j+1} \ldots u_k) \in F$ and hence $u_0 u_1 \ldots u_{i-1} (u_i u_{i+1} \ldots u_{j-1})^h$ $u_j u_{j+1} \ldots u_k \in L$ for all h. ∎

Example 2.10. Let L be the language of all strings over $\{0, 1, 2\}$ which contains an even number of 0s. Then the pumping-condition of Theorem 2.9 is satisfied with parameter $n = 3$: Given $u_0 u_1 u_2 u_3 \in L$, there are three cases:

- u_1 contains an even number of 0s. Then removing u_1 from the word or inserting it arbitrarily often does not make the number of 0s in the word odd; hence $u_0(u_1)^* u_2 u_3 \subseteq L$.
- u_2 contains an even number of 0s. Then $u_0 u_1 (u_2)^* u_3 \subseteq L$.
- u_1 and u_2 contain both an odd number of 0s. Then $u_1 u_2$ contains an even number of 0s and $u_0(u_1 u_2)^* u_3 \subseteq L$.

Hence the pumping condition is satisfied for L.

Let H be the language of all words which contain a different number of 0s and 1s. Let k be any constant. Now let $u_0 = 0, u_1 = 0, \ldots, u_{k-1} = 0, u_k = 1^{k+k!}$. If the pumping condition would be satisfied for H then there are i, j with $0 < i < j \leq k$ and

$$0^i (0^{j-i})^* 0^{k-j} 1^{k+k!} \subseteq H.$$

So fix this i, j and take $h = \frac{k!}{j-i} + 1$ (which is a natural number). Now one sees that $0^i 0^{(j-i)h} 0^{k-j} 1^{k+k!} = 0^{k+k!} 1^{k+k!} \notin H$, hence the pumping condition is not satisfied.

Theorem 2.11: Ehrenfeucht, Parikh and Rozenberg [22]. *A language L is regular if and only if both L and its complement satisfy the block pumping lemma.*

Proof. The proof is based on showing the even more restrictive block cancellation property. That is, it is shown that L is regular iff there is a constant $k \geq 3$ such that the following condition holds for k:

> (E_k): for all words $u_0, u_1, \ldots, u_k$, there are $i, j \in \{0, 1, \ldots, k-1\}$ with $i < j$ and $L(u_0 \ldots u_k) = L(u_0 \ldots u_i \cdot u_{j+1} \ldots u_k)$.

This says in particular, if one cuts a word into $k + 1$ blocks with the zeroth and the last block possibly empty then one can find an interval of some blocks not containing the zeroth and the last such that deleting the interval from the word does not change membership in L, so if x is a member of L so is the shorter word and if x is a member of the complement of L so again is the shorter word.

On one hand, it will be shown that every regular set satisfies (E_k) for some k and on the other hand that whenever a set satisfies (E_k) for some k then it is regular. Furthermore, for each k and for each fixed alphabet, there are only finitely many sets satisfying (E_k).

That regular sets satisfy (E_k) comes directly out of analysing the states of a dfa recognising the corresponding regular language with k states and by choosing i, j such that there is the same state after reading $u_0 \ldots u_i$ and after reading $u_0 \ldots u_j$.

For the other direction, one has to choose a constant $c > k$ such that every two-colouring of pairs (i, j) from $\{0, 1, \ldots, c\}$ has a homogeneous set of size $k + 1$; this constant exists by the finite version of Ramsey's Theorem of Pairs [62].

Ramsey's Theorem of pairs says the following: For each k there is a c such that if one assigns to each pair (i, j) with $i < j$ and $i, j \in \{0, 1, \ldots, c-1\}$ one of the colours white or red then there is a subset $\{h_0, h_1, \ldots, h_k\}$ of $\{0, 1, \ldots, c\}$ such that all pairs $(i, j), (i', j')$ with $i < j$ and $i' < j'$ and $i, j, i', j' \in \{h_0, h_1, \ldots, h_{k-1}\}$ have the same colour. Such a subset is called homogeneous.

Ramsey's Theorem of Pairs has been a very useful tool in proving combinatorial properties in many branches of mathematics including the block pumping lemma.

Now let H_1, H_2 be two sets which satisfy (E_k) and assume they are identical on all strings of length up to c. Now assume by way of contradiction that $H_1 \neq H_2$.

Let x be the length-lexicographically first string on which $H_1(x) \neq H_2(x)$ and let u_0 be ε, u_h be the h-th symbol of x for $h = 1, \ldots, c-1$ and u_c is the remaining part of the word x. Furthermore, for $i, j \in \{0, 1, \ldots, c\}$ with $i < j$, make a two-colouring col such that the following holds: If $H_1(u_0 u_1 \ldots u_i \cdot u_{j+1} u_{j+2} \ldots u_c) = H_1(x)$ then $col(i, j) = white$ else $col(i, j) = red$.

By Ramsey's Theorem of Pairs there are $h_0, h_1, \ldots, h_k$ on which col is homogeneous and one can consider the splitting of x into $k + 1$ blocks $u_0 \ldots u_{h_0}$, $u_{h_0+1} \ldots u_{h_1}$, …, $u_{h_{k-1}+1} \ldots u_c$. These splittings again satisfy the property (E_k) for H_1. As there must be $i, j \in \{h_0, h_1, \ldots, h_{k-1}\}$ with $i < j$ and $H_1(u_0 u_1 \ldots u_{h_i} \cdot u_{h_j+1} u_{h_j+2} \ldots u_c) = H_1(x)$, the homogeneous colour is white.

Furthermore, there must, by (E_k) for H_2, exist $i', j' \in \{h_0, h_1, \ldots, h_{k-1}\}$ with $i' < j'$ and $H_2(u_1 u_2 \ldots u_{i'} \cdot u_{j'+1} u_{j'+2} \ldots u_{c'}) = H_2(x)$. Due to homogenicity, it also holds that $H_1(u_1 u_2 \ldots u_{i'} \cdot u_{j'+1} u_{j'+2} \ldots u_{c'}) = H_1(x)$. On one hand, this gives $H_1(u_1 u_2 \ldots u_{i'} \cdot u_{j'+1} u_{j'+2} \ldots u_{c'}) \neq H_2(u_1 u_2 \ldots u_{i'} \cdot u_{j'+1} u_{j'+2} \ldots u_{c'})$, on the other hand the choice of x gives that H_1, H_2 coincide on this string as it is shorter than x. This contradiction leads to the conclusion that H_1 and H_2 coincide on all strings whenever both satisfy (E_k) and $H_1(y) = H_2(y)$ for all strings up to length c. So, whenever two sets satisfy (E_k) and when they coincide on strings up to length c then they are equal.

Note that when L satisfies (E_k), so do also all derivatives $L_x = \{y : xy \in L\}$: If $\tilde{u}_0, \tilde{u}_1, \ldots, \tilde{u}_k$ are $k + 1$ strings then one considers $x\tilde{u}_0, \tilde{u}_1, \ldots, \tilde{u}_k$ for L and selects indices $i, j \in \{0, 1, \ldots, k-1\}$ with $i < j$ such that $L(x\tilde{u}_0\tilde{u}_1 \ldots \tilde{u}_k) = L(x\tilde{u}_0 \ldots \tilde{u}_i \cdot \tilde{u}_{j+1} \ldots \tilde{u}_k)$. It follows that $L_x(\tilde{u}_0\tilde{u}_1 \ldots \tilde{u}_k) = L_x(\tilde{u}_0 \ldots \tilde{u}_i \cdot \tilde{u}_{j+1} \ldots \tilde{u}_k)$ and hence also L_x satisfies (E_k).

Each derivative L_x is determined by the values $L_x(y)$ for the y with $|y| \leq c$. So there are at most $2^{1+d+d^2+\ldots+d^c}$ many derivatives where d is the number of symbols in the alphabet; in particular there are only finitely many derivatives. The language L is regular by the Theorem of Myhill and Nerode (Theorem 2.19). ∎

However, there are non-regular languages L which satisfy the block pumping lemma. Morse as well as Thue [75] constructed an infinite binary sequence in which there is no non-empty subword of the form www. This sequence witnesses that there are cubefree strings of arbitrary length and this fact is used to construct nonregular set L satisfying the block pumping lemma.

Theorem 2.12: Sequence of Morse and Thue [75]. *Let $a_0 = 0$ and, for all n, $a_{2n} = a_n$ and $a_{2n+1} = 1 - a_n$. Then the infinite binary sequence $a_0 a_1 \ldots$ does not contain a subword of the form www.*

Proof. In the following, call a word a "cube" if it is not empty and of the form www for some string w.

Assume by way of contradiction that the sequence of Morse and Thue contains a cube as a subword and let www be the first such subword of the sequence. Let k be the length of w and $w_1 w_2 \ldots w_k$ be the symbols in w (in this order).

In the case that w has even length, then consider the first position $2n + m$ with $m \in \{0, 1\}$ of www in the sequence. If $m = 0$ then $a_n = a_{2n+m}, a_{n+1} = a_{2n+2+m}, \ldots, a_{n+3k/2} = a_{2n+3k}$ else $a_n = 1 - a_{2n+m+1}, a_{n+1} = 1 - a_{2n+2+m+1}, \ldots, a_{n+3k/2} = 1 - a_{2n+3k+1}$. In both cases, $a_n a_{n+1} \ldots a_{n+3k/2}$ is of the form vvv where v has the length $k/2$ and occurs before www. As www was chosen to be the first cube occurring in the sequence, this case does not apply and k must be odd.

For the case of an odd k and for each $h \in \{1, 2, \ldots, k - 1\}$, either the first or the second occurrence of w satisfies that w_h is at a position of the form $2n$ and w_{h+1} at a position of the form $2n+1$ so that, by the construction of the sequence, $w_{h+1} = 1 - w_h$. Furthermore, by the same principle applied to the position where one copy of w ends and the next starts, one has that $w_1 = 1 - w_k$. However, as w has odd length, one also has $w_1 = w_k$; for example if w has length 5 then w is either 01010 or 10101. This gives a contradiction and therefore this case does also not occur. Hence the sequence of Morse and Thue has no subword which is a cube. ∎

Theorem 2.13 [12]. *There is a block pumpable language which is not regular.*

Proof. Let L contain all words which either contain a cube or whose length is not a power of 10, so nonmembers of L have one of the lengths $1, 10, 100, 1000, \ldots$ and no other length. Now one shows that L has the block pumping constant 5. Assume that $w \in L$ and w is split into blocks $u_0, u_1, u_2, u_3, u_4, u_5$ and assume that u_1, u_2, u_3, u_4 are all non-empty, as if one of them is empty one can pump that empty block. Now it is shown that one can select one of the possible pumps u_1, $u_1 u_2$, u_3, $u_3 u_4$ such that when omitting or repeating an arbitrary time the selected pump in w, the so modified word is again in L. In other words, one of the following languages is a subset of L: $u_0(u_1)^*u_2u_3u_4u_5$, $u_0(u_1u_2)^*u_3u_4u_5$, $u_0u_1u_2(u_3)^*u_4u_5$ and $u_0u_1u_2(u_3u_4)^*u_5$.

First consider the case that $|u_1u_2| \leq |u_3u_4|$. In this case, $|u_0u_1u_2u_1u_2u_3u_4u_5| \leq |u_0u_3u_4u_5| \cdot 3$ and only one of the words $u_0u_3u_4u_5$, $u_0u_2u_3u_4u_5$, $u_0(u_1)^2u_2u_3u_4u_5$ and $u_0(u_1u_2)^2u_3u_4u_5$ has a length which is a power of 10. Hence one can select the pump to be either u_1 or u_1u_2 such that when the pump is omitted or doubled the resulting word does not have a length which is a power of 10 and is therefore in L. Furthermore, for both possible pumps and $h \geq 3$, the words $u_0(u_1)^hu_2u_3u_4u_5$ and $u_0(u_1u_2)^hu_3u_4u_5$ do both contain a cube and are in L. Thus, one can choose the pump such that all pumped words are in L.

Second in the case that $|u_3u_4| < |u_1u_2|$, one can do the same proof as before, only with the possible pumps being u_3 and u_3u_4, one of them works.

To see that L is not regular, note that for each power of 10 there is a word in the complement of L which consists of the corresponding first symbols of the sequence of Morse and Thue. Note that the complement of L is now infinite but cannot satisfy any pumping lemma as it contains only cubefree words. Thus the complement of L and, hence, also L itself cannot be regular. ∎

Quiz 2.14. *Which of the following languages over $\Sigma = \{0,1,2,3\}$ satisfy the pumping-condition from Theorem 2.9:*
(a) $\{00, 111, 22222\}^* \cap \{11, 222, 00000\}^* \cap \{22, 000, 11111\}^*$,
(b) $\{0^i1^j2^k : i + j = k + 5555\}$,
(c) $\{0^i1^j2^k : i + j + k = 5555\}$,
(d) $\{w : w \text{ contains more } 1 \text{ than } 0\}$*?*

Exercise 2.15. *Find the optimal constants for the Block Pumping Lemma for the following languages:*
(a) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* :$ *at least one nonzero digit a occurs in w at least three times*$\}$*;*
(b) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* : |w| = 255\}$*;*
(c) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* :$ *the length* $|w|$ *is not a multiple of* $6\}$*;*
Here the constant for a language L is the least n such that for all words $u_0, u_1, \ldots, u_n$ the implication

$$u_0u_1u_2 \ldots u_n \in L \Rightarrow \exists i, j\, [0 < i < j \leq n \text{ and } u_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^*u_j \ldots u_n \subseteq L]$$

holds.

Exercise 2.16. *Find the optimal constants for the Block Pumping Lemma for the following languages:*

(a) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* : w$ *is a multiple of* $25\}$;
(b) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* : w$ *is not a multiple of* $3\}$;
(c) $\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* : w$ *is a multiple of* $400\}$.

Exercise 2.17. *Find a regular language L so that the constant of the Block Pumping Lemma for L is 4 and for the complement of L is 4196.*

Exercise 2.18. *Give an example L of a language which satisfies Theorem 1.35 (a) (where for every $w \in L$ of length at least k there is a splitting $xyz = w$ with $|xy| \leq k$, $|y| > 0$ and $xy^*z \subseteq L$) but does not satisfy Theorem 2.9 (the Block Pumping Lemma).*

Theorem 2.19: Myhill and Nerode's Minimal DFA [56]**.** *Given a language L, let $L_x = \{y \in \Sigma^* : xy \in L\}$ be the derivative of L to x. The language L is regular iff the number of different derivatives L_x is finite; furthermore, for languages with exactly n derivatives, one can construct a complete dfa having n and there is no complete dfa with less than n states which recognises L.*

Proof. Let $(Q, \Sigma, \delta, s, F)$ be a deterministic finite automaton recognising L. If $\delta(s,x) = \delta(s,y)$ then for all $z \in \Sigma^*$ it holds that $z \in L_x$ iff $\delta(\delta(s,x),z) \in F$ iff $\delta(\delta(s,y),z) \in F$ iff $z \in L_y$. Hence the number of different sets of the form L_x is a lower bound for the size of the states of the dfa.

Furthermore, one can directly build the dfa by letting $Q = \{L_x : x \in \Sigma^*\}$ and define for $L_x \in Q$ and $a \in \Sigma$ that $\delta(L_x, a)$ is the set L_{xa}. The starting-state is the set L_ε and $F = \{L_x : x \in \Sigma^* \wedge \varepsilon \in L_x\}$.

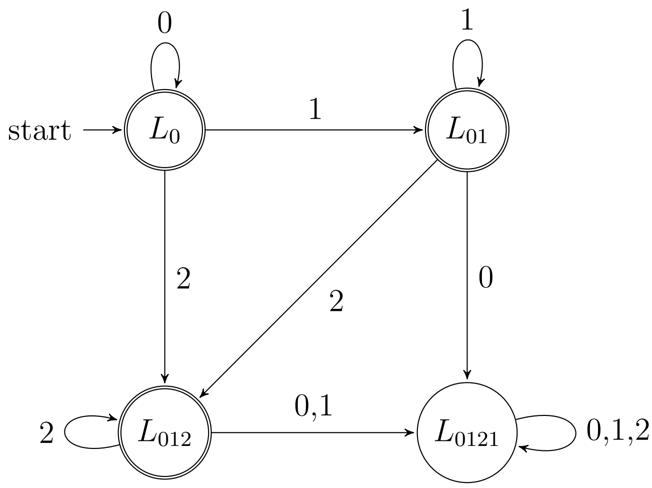
In practice, one would of course pick representatives for each state, so there is a finite subset Q of Σ^* with $\varepsilon \in Q$ and for each set L_y there is exactly one $x \in Q$ with $L_x = L_y$. Then $\delta(x,a)$ is that unique y with $L_y = L_{xa}$.

For the verification, note that there are only finitely many different derivatives, so the set Q is finite. Furthermore, each state can be reached: For $x \in Q$, one can reach the state x by feeding the word x into the automaton. Assume now that $L_x = L_y$. Then $L_{xa} = \{z : xaz \in L\} = \{z : az \in L_x\} = \{z : az \in L_y\} = \{z : yaz \in L\} = L_{ya}$, thus the transition function δ is indeed independent of whether x or y is chosen to represent L_x and will select the unique member z of Q with $L_z = L_{xa} = L_{ya}$. In addition, the rule for making exactly the states x with $\varepsilon \in L_x$ be accepting is correct: The reason is that, for $x \in Q$, the automaton is in state x after reading x and x has to be accepted by the automaton iff $x \in L$ iff $\varepsilon \in L_x$. ∎

In the case that some derivative is $\emptyset$, one can get an automaton which has one less state if one decides not to represent $\emptyset$; the resulting dfa would then be incomplete, that is, there would be nodes q and symbols a with $\delta(q,a)$ being undefined; if the automaton ends up in this situation, it would just reject the input without further analysis. An incomplete dfa is a variant of a dfa which is still very near to a complete dfa but has already gone a tiny step in direction of an nfa (as defined in Description 2.38 below).

Remark 2.20. Although the above theorem is published by Anil Nerode [56], it is general known as the Theorem of Myhill and Nerode and both scientists, John Myhill and Anil Nerode, are today acknowledged for this discovery. The notion of a derivative was fully investigated by Brzozowski when working on regular expressions [7].

Example 2.21. If $L = 0^*1^*2^*$ then $L_0 = 0^*1^*2^*$, $L_{01} = 1^*2^*$, $L_{012} = 2^*$ and $L_{0121} = \emptyset$. Every further L_x is equivalent to one of these four: If $x \in 0^*$ then $L_x = L$; if $x \in 0^*1^+$ then $L_x = 1^*2^*$ as a 0 following a 1 makes the word to be outside L; if $x \in 0^*1^*2^+$ then $L_x \in 2^*$. If $x \notin 0^*1^*2^*$ then also all extensions of x are outside L and $L_x = \emptyset$. The automaton obtained by the construction of Myhill and Nerode is the following.



As $L_{0121} = \emptyset$, one could also omit this node and would get an incomplete dfa with all states being accepting. Then a word is accepted as long as one can go on in the automaton on its symbols.

Example 2.22. Consider the language $\{0^n1^n : n \in \mathbb{N}\}$. Then $L_{0^n} = \{0^m1^{m+n} : m \in \mathbb{N}\}$ is unique for each $n \in \mathbb{N}$. Hence, if this language would be recognised by a dfa, then the dfa would need infinitely many states, what is impossible.

Lemma 2.23: Jaffe's Matching Pumping Lemma [37]. A language $L \subseteq \Sigma^*$ is regular iff there is a constant k such that for all $x \in \Sigma^*$ and $y \in \Sigma^k$ there are u, v, w with $y = uvw$ and $v \neq \varepsilon$ such that, for all $h \in \mathbb{N}$, $L_{xuv^hw} = L_{xy}$.

Proof. Assume that L satisfies Jaffe's Matching Pumping Lemma with constant k. For every word z with $|z| \geq k$ there is a splitting of z into xy with $|y| = k$. Now there is a shorter word xuw with $L_{xuw} = L_{xy}$; thus one can find, by repeatingly using this argument, that every derivative L_z is equal to some derivative $L_{z'}$ with $|z'| < k$. Hence there are only $1 + |\Sigma| + \ldots + |\Sigma|^{k-1}$ many different derivatives and therefore the language is regular by the Theorem of Myhill and Nerode.

The converse direction follows by considering a dfa recognising L and letting k be larger than the number of states in the dfa. Then when the dfa processes a word xyz and $|y| = k$, then there is a splitting of y into uvw with $v \neq \varepsilon$ such that the dfa is in the same state when processing xu and xuv. It follows that the dfa is, for every h, in the same state when processing xuv^h and therefore it accepts xuv^hwz iff it accepts xyz. Thus $L_{xuv^hw} = L_{xy}$ for all h. ∎

Exercise 2.24. *Assume that the alphabet Σ has 5000 elements. Define a language $L \subseteq \Sigma^*$ such that Jaffe's Matching Pumping Lemma is satisfied with constant $k = 3$ while every deterministic finite automaton recognising L has more than 5000 states. Prove the answer.*

Exercise 2.25. *Find a language which needs for Jaffe's Matching Pumping Lemma at least constant $k = 100$ and can be recognised by a deterministic finite automaton with 100 states. Prove the answer.*

Consider the following weaker version of Jaffe's Pumping Lemma which follows from it.

Corollary 2.26. *Regular languages L and also some others satisfy the following condition:*

There is a constant k such that for all $x \in \Sigma^*$ and $y \in \Sigma^k$ with $xy \in L$ there are u, v, w with $y = uvw$ and $v \neq \varepsilon$ such that, for all $h \in \mathbb{N}$, $L_{xuv^hw} = L_{xy}$.

That is, in Corollary 2.26, one postulates the property of Jaffe's Pumping Lemma only for members of L. Then it loses its strength and is no longer matching.

Exercise 2.27. *Show that the language $L = \{\varepsilon\} \cup \{0^n1^m2^k3 : n = m \text{ or } k = 0\}$ is a context-free language which satisfies Corollary 2.26 but is not regular. Furthermore, show directly that this language does not satisfy Jaffe's Pumping Lemma itself; this is expected, as only regular languages satisfy it.*

Exercise 2.28. *Is the following statement true: If L satisfies Corollary 2.26 and H is regular then $L \cdot H$ satisfies Corollary 2.26?*

Exercise 2.29. *Call a language prefix-free if whenever $vw \in L$ and $w \neq \varepsilon$ then $v \notin L$. Does every prefix-free language L for which L^{mi} satisfies Theorem 1.35 (a) also satisfy Corollary 2.26? Prove the answer.*

Exercise 2.30. *Let $\Sigma = \{0, 1, 2\}$. Call a word v square-containing iff it has a non-empty subword of the form ww with $w \in \Sigma^+$ and let L be the language of all square-containing words; call a word v palindrome-containing iff it has a non-empty subword of the form ww^{mi} or waw^{mi} with $a \in \Sigma$ and $w \in \Sigma^+$ and let H be the language of all palindrome-containing words.*

Determine the exact levels of L and H in the Chomsky hierarchy. Which of the pumping lemmas (except for the block pumping lemma) do they satisfy? If they are regular, provide a dfa.

The overall goal of Myhill and Nerode was also to provide an algorithm to compute for a given complete dfa a minimal complete dfa recognising the same language.

Algorithm 2.31: Myhill's and Nerodes Algorithm to Minimise Deterministic Finite Automata [56]**.**
Given: Complete dfa $(Q, \Sigma, \delta, s, F)$.

Computing Set R of Reachable States:
Let $R = \{s\}$;
While there is $q \in R$ and $a \in \Sigma$ with $\delta(q, a) \notin R$, let $R = R \cup \{\delta(q, a)\}$.

Identifying When States Are Distinct:
Make a relation $\gamma \subseteq R \times R$ which contains all pairs of states (q, p) such that the automaton behaves differently when starting from p or from q;
Initialise γ as the set of all $(p, q) \in R \times R$ such that exactly one of p, q is accepting;
While there are $(p, q) \in R \times R$ and $a \in \Sigma$ such that $(p, q) \notin \gamma$ and $(\delta(p, a), \delta(q, a)) \in \gamma$, put $(p, q), (q, p)$ into γ.

Building Minimal Automaton:
Let $Q' = \{q \in R$ such that all $p \in R$ with $p < q$ (according to some default ordering of Q) satisfy $(p, q) \in \gamma\}$;
Let s' be the unique state in Q' such that $(s, s') \notin \gamma$;
For $p \in Q'$ and $a \in \Sigma$, let $\delta'(p, a)$ be the unique $q \in Q'$ such that $(q, \delta(p, a)) \notin \gamma$;
Let $F' = Q' \cap F$;
Now $(Q', \Sigma, \delta', s', F')$ is the minimal automaton to be constructed.

Verification. First one should verify that R contains exactly the reachable states. Clearly s is reachable by feeding ε into the automaton. By induction, when $\delta(q, a)$

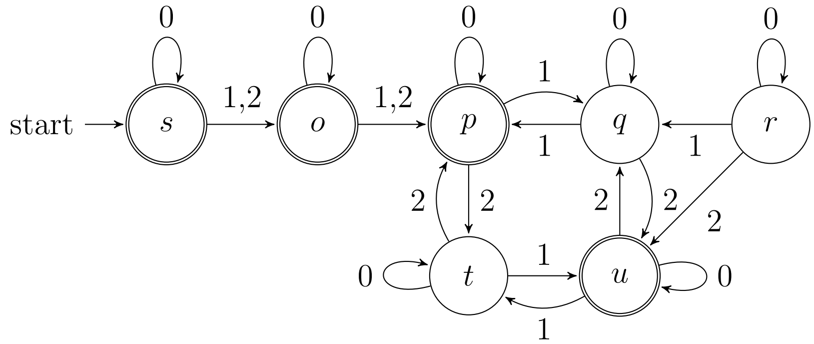
is added to the set R then q is reachable, by some word x; it follows that $\delta(q,a)$ is reachable by the word xa. Furthermore, the adding of nodes is repeated until the set R is closed, that is, for all $q \in R$ and $a \in \Sigma$ the state $\delta(q,a)$ is also in R. Thus one cannot reach from any state inside the final R a state outside the final R and therefore the final R consists exactly of the reachable states.

Second one verifies that the final version of γ contains exactly the pairs $(p,q) \in R \times R$ such that when starting from p or q, the behaviour of the automaton is different. When (p,q) are put into γ at the initialisation then $\delta(p,\varepsilon), \delta(q,\varepsilon)$ differ in the sense that one ends up in a rejecting and one ends up in an accepting state, that is, ε witnesses that p,q are states of different behaviour. Now one verifies that this invariance is kept for the inductive step: When $(\delta(p,a),\delta(q,a)) \in \gamma$ and (p,q) are going to be added into γ then there is by induction hypothesis a y such that exactly one of $\delta(\delta(p,a),y), \delta(\delta(q,a),y)$ is an accepting state, these two states are equal to $\delta(p,ay), \delta(q,ay)$ and therefore ay witnesses that p,q are states of different behaviour.

The next part of the verification is to show that γ indeed captures all these of states in R of different behaviour. So assume that $y = a_1 a_2 \ldots a_n$ witnesses that when starting at p the automaton accepts y and when starting with q then the automaton rejects y. Thus $(\delta(p,y),\delta(q,y)) \in \gamma$. Now one shows by induction for $m = n-1, n-2, \ldots, 0$ that $(\delta(p,a_1 a_2 \ldots a_m), \delta(q, a_1 a_2 \ldots a_m))$ goes eventually into γ: by induction hypothesis $(\delta(p, a_1 a_2 \ldots a_m a_{m+1}), \delta(q, a_1 a_2 \ldots a_m a_{m+1}))$ is at some point of time going into γ and therefore the pair $(\delta(p, a_1 a_2 \ldots a_m), \delta(q, a_1 a_2 \ldots a_m))$ satisfies that, when applying the symbol a_{m+1} to the two states, the resulting pair is in γ, hence $(\delta(p, a_1 a_2 \ldots a_m), \delta(q, a_1 a_2 \ldots a_m))$ will eventually qualify in the search condition and therefore at some time point go into γ. It follows that this also holds for all m down to 0 by the induction and that $(p,q),(q,p)$ go into γ. Thus all pairs of states of distinct behaviour in $R \times R$ go eventually into γ.

Now let $<$ be the linear order on the states of $<$ which is used by the algorithm. If for a state p there is a state $q < p$ with $(p,q) \notin \gamma$ then the state q has the same behaviour as p and is redundant; therefore one picks for Q' all those states for which there is no smaller state of the same behaviour. Note that (p,p) never goes into γ for any $p \in R$ and therefore for each p there is a smallest q such that $(p,q) \notin \gamma$ and for each p there is a $q \in R'$ with the same behaviour. In particular s' exists. Furthermore, one can show by induction for all words that $\delta(s,w)$ is an accepting state iff $\delta'(s',w)$ is one. A more general result will be shown: The behaviour of $\delta(s,w)$ and $\delta'(s',w)$ are not different, that is, $(\delta(s,w), \delta'(s',w)) \notin \gamma$. Clearly $(\delta(s,\varepsilon), \delta'(s',\varepsilon)) \notin \gamma$. Now, for the inductive step, assume that $(\delta(s,w), \delta'(s',w)) \notin \gamma$ and $a \in \Sigma$. Now $(\delta(\delta(s,w),a), \delta(\delta'(s',w),a)) \notin \gamma$, that is, have the same behaviour. Furthermore, by the definition of δ', $(\delta(\delta'(s,w),a), \delta'(\delta'(s',w),a)) \notin \gamma$, that is, also have the same behaviour. Now $(\delta(\delta(s,w),a), \delta'(\delta'(s',w),a)) \notin \gamma$, as $\delta(\delta(s,w),a)$ has the same behaviour as $\delta(\delta'(s', w), a)$ and $\delta(\delta'(s', w), a)$ has the same behaviour as $\delta'(\delta'(s', w), a)$. So the new minimal automaton has the same behaviour as the original automaton. ▮

Exercise 2.32. Let the following deterministic finite automaton be given:



Make an equivalent minimal complete dfa using the algorithm of Myhill and Nerode.

Exercise 2.33. *Assume that the alphabet is* $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ *and the set of states is* $\{(a, b, c) : a, b, c \in \Sigma\}$. *Furthermore assume the transition function* δ *is given by* $\delta((a, b, c), d) = (b, c, d)$ *for all* $a, b, c, d \in \Sigma$, *the starting state is* $(0, 0, 0)$ *and that the set of final states is* $\{(1, 1, 0), (3, 1, 0), (5, 1, 0), (7, 1, 0), (9, 1, 0)\}$.

This dfa has 1000 *states. Find a smaller dfa for this set and try to get the dfa as small as possible.*

Exercise 2.34. *Assume that the alphabet is* $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ *and the set of states is* $\{(a, b, c) : a, b, c \in \Sigma\}$. *Furthermore assume the transition function* δ *is given by* $\delta((a, b, c), d) = (b, c, d)$ *for all* $a, b, c, d \in \Sigma$, *the starting state is* $(0, 0, 0)$ *and that the set of final states is* $\{(1, 2, 5), (3, 7, 5), (6, 2, 5), (8, 7, 5)\}$.

This dfa has 1000 *states. Find a smaller dfa for this set and try to get the dfa as small as possible.*

Exercise 2.35. *Consider the following context-free grammar:*

$(\{S, T, U\}, \{0, 1, 2, 3\}, P, S)$ *with* $P =$
$\{S \to TTT|TTU|TUU|UUU,\ T \to 0T|T1|01,\ U \to 2U|U3|23\}$.

The language L *generated by the grammar is regular. Provide a dfa with the minimal number of states recognising* L.

Exercise 2.36. *Consider the following context-free grammar:*

$(\{S, T, U\}, \{0, 1, 2, 3, 4, 5\}, P, S)$ *with* $P =$
$\{S \to TS|SU|T23U,\ T \to 0T|T1|01,\ U \to 4U|U5|45\}$.

The language L generated by the grammar is regular. Provide a dfa with the minimal number of states recognising L, the dfa does not need to be complete. Furthermore, provide a regular expression for the language L.

There are quite simple tasks where an automaton to check this might become much larger than it is adequate for the case. For example, to check whether a string contains a symbol twice, one would guess which symbol is twice and then just verify that it occurs twice; however, a deterministic finite automaton cannot do it and the following example provides a precise justification. Therefore, this chapter will look into mechanisms to formalise this intuitive approach which is to look at a word like 0120547869 where one, by just looking at it, might intuitively see that the 0 is double and then verify it with a closer look. Such type of intuition is not possible to a deterministic finite automaton; however, non-determinism permits to model intuitive decisions as long as their is a way to make sure that the intuitive insight is correct (like scanning the word for the twice occurring letter).

Example 2.37. Assume that Σ has n elements. Consider the set L of all strings which contain at least one symbol at least twice.

There are at least $2^n + 1$ sets of the form L_x: If $x \in L$ then $L_x = \Sigma^*$ else $\varepsilon \notin L_x$. Furthermore, for $x \notin L$, $\Sigma \cap L_x = \{a \in \Sigma : a \text{ occurs in } x\}$. As there are 2^n subsets of Σ, one directly gets that there are 2^n states of this type.

On the other hand, one can also see that $2^n + 1$ is an upper bound. If the dfa has not seen any symbol twice so far then it just has to remember which symbols it has seen else the automaton needs just one additional state to go when it has seen some symbol twice. Representing the first states by the corresponding subsets of Σ and the second state by the special symbol #, the dfa would has the following parameters: $Q = Pow(\Sigma) \cup \{\#\}$, Σ is the alphabet, $\emptyset$ is the starting state and # is the unique final state. Furthermore, δ is is given by three cases: if $A \subseteq \Sigma$ and $a \in \Sigma - A$ then $\delta(A, a) = A \cup \{a\}$, if $A \subseteq \Sigma$ and $a \in A$ then $\delta(A, a) = \#$, $\delta(\#, a) = \#$.

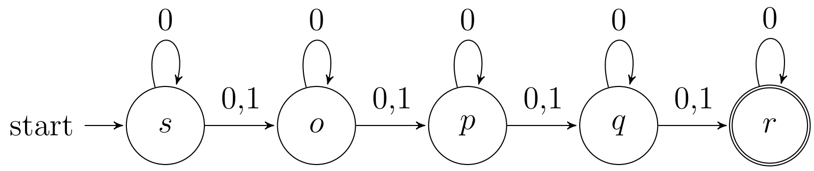
Description 2.38: Non-Deterministic Finite Automaton. A non-deterministic automaton can guess information and, in the case that it guessed right, verify that a word is accepting.

A non-deterministic automaton $(Q, \Sigma, \delta, s, F)$ differs from the deterministic automaton in the way that δ is a multi-valued function, that is, for each $q \in Q$ and $a \in \Sigma$ the value $\delta(q, a)$ is a set of states.

Now one defines the acceptance-condition using the notion of a run: One says a string $q_0 q_1 \ldots q_n \in Q^{n+1}$ is a run of the automaton on input $a_1 \ldots a_n$ iff $q_0 = s$ and $q_{m+1} \in \delta(q_m, a_{m+1})$ for all $m \in \{1, \ldots, n\}$; note that the run has one symbol more than the string processed. The non-deterministic automaton accepts a word w iff there is a run on the input w whose last state is accepting.

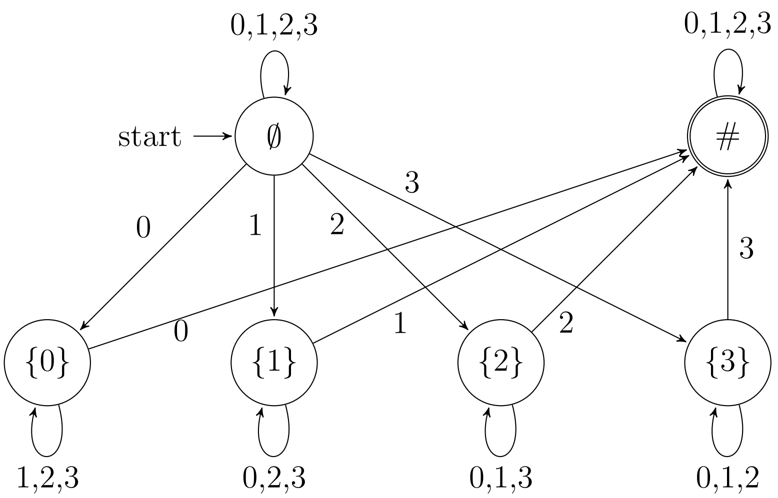
Note that for accepting a word, there needs only to be at least one accepting run; other rejecting runs might also exist. For rejecting a word, all runs which exist must be rejecting, this includes the case that there is no run at all (neither an accepting nor a rejecting).

Example 2.39. Consider the following non-deterministic automaton which accepts all words which have at least four letters and at most four 1's.



On input 00111, accepting runs are $s\,s\,o\,p\,q\,r$ and $s\,o\,o\,p\,q\,r$; on input 11111 there is no accepting run, as the automaton has to advance $s\,o\,p\,q\,r$ and then, on the last input 1, gets stuck as it cannot move. The input 000 has no accepting run, as the run $s\,o\,p\,q$ does not reach the final accepting state r and all other runs end up in one of the states s, o, p without even reaching q. Thus 00111 is accepted and $11111, 000$ are rejected by this nfa.

Example 2.40: Large DFA and small NFA. For the dfa with $2^n + 1$ states from Example 2.37, one can make an nfa with $n + 2$ states (here for $n = 4$ and $\Sigma = \{0, 1, 2, 3\}$). Thus an nfa can be exponentially smaller than a corresponding dfa.



In general, Q contains $\emptyset$ and $\{a\}$ for all $a \in \Sigma$ and $\#$; $\delta(\emptyset, a) = \{\emptyset, \{a\}\}$; $\delta(\{a\}, b)$ is $\{a\}$ in the case $a \neq b$ and is $\#$ in the case $a = b$; $\delta(\#, a) = \#$; $\emptyset$ is the starting state;

is the only accepting state.

So the nfa has $n+2$ and the dfa has 2^n+1 states (which cannot be made better). So the actual size of the dfa is more than a quarter of the theoretical upper bound 2^{n+2} which will be given by the construction found by Büchi [8, 9] as well as Rabin and Scott [61]. Their general construction which permits to show that every nfa with n states is equivalent to a dfa with 2^n states, that is, the nfa and the dfa constructed recognise the same language.

Theorem 2.41: Determinisation of NFAs [8, 9, 61]**.** *For each nfa* $(Q, \Sigma, \delta, s, F)$ *with* $n = |Q|$ *states, there is an equivalent dfa whose* 2^n *states are the subsets* Q' *of* Q*, whose starting state is* $\{s\}$*, whose update-function* δ' *is given by* $\delta'(Q', a) = \{q'' \in Q : \exists q' \in Q' \, [q'' \in \delta(q', a)]\}$ *and whose set of accepting states is* $F' = \{Q' \subseteq Q : Q' \cap F \neq \emptyset\}$*.*

Proof. It is clear that the automaton defined in the statement of the theorem is a dfa: For each set $Q' \subseteq Q$ and each $a \in \Sigma$, the function δ' selects a unique successor $Q'' = \delta'(Q', a)$. Note that Q'' can be the empty set and that, by the definition of δ', $\delta'(\emptyset, a) = \emptyset$.

Assume now that the nfa accepts a word $w = a_1 a_2 \ldots a_m$ of m letters. Then there is an accepting run $(q_0, q_1, \ldots, q_m)$ on this word with $q_0 = s$ and $q_m \in F$. Let $Q_0 = \{s\}$ be the starting state of the dfa and, inductively, $Q_{k+1} = \delta'(Q_k, a_{k+1})$ for $k = 0, 1, \ldots, m-1$. One can verify by induction that $q_k \in Q_k$ for all $k \in \{0, 1, \ldots, m\}$: This is true for $q_0 = s$ by definition of Q_0; for the inductive step, if $q_k \in Q_k$ and $k < m$, then $q_{k+1} \in \delta(q_k, a_{k+1})$ and therefore $q_{k+1} \in Q_{k+1} = \delta'(Q_k, a_{k+1})$. Thus $Q_m \cap F$ contains the element q_m and therefore Q_m is an accepting state in the dfa.

For the converse direction on a given word $w = a_1 a_2 \ldots a_m$, assume that the run $(Q_0, Q_1, \ldots, Q_m)$ of the dfa on this word is accepting. Thus there is $q_m \in Q_m \cap F$. Now one can, inductively for $k = m-1, m-2, \ldots, 2, 1, 0$ choose a q_k such that $q_{k+1} \in \delta(q_k, a_{k+1})$ by the definition of δ'. It follows that $q_0 \in Q_0$ and therefore $q_0 = s$. Thus the so defined sequence $(q_0, q_1, \ldots, q_m)$ is an accepting run of the nfa on the word w and the nfa accepts the word w as well.

This shows that the dfa is equivalent to the nfa, that is, it accepts and it rejects the same words. Furthermore, as an n-element set has 2^n subsets, the dfa has 2^n states. ∎

Note that this construction produces, in many cases, too many states. Thus one would consider only those states (subsets of Q) which are reached from others previously constructed; in some cases this can save a lot of work. Furthermore, once the dfa is constructed, one can run the algorithm of Myhill and Nerode to make a minimal dfa out of the constructed one.

Example 2.42. Consider the nfa $(\{s,q\},\{0,1\},\delta,s,\{q\})$ with $\delta(s,0) = \{s,q\}$, $\delta(s,1) = \{s\}$ and $\delta(q,a) = \emptyset$ for all $a \in \{0,1\}$.

Then the corresponding dfa has the four states $\emptyset, \{s\}, \{q\}, \{s,q\}$ where $\{q\}, \{s,q\}$ are the final states and $\{s\}$ is the initial state. The transition function δ' of the dfa is given as

$$\begin{aligned}
&\delta'(\emptyset,a) = \emptyset \text{ for } a \in \{0,1\},\\
&\delta'(\{s\},0) = \{s,q\},\ \delta'(\{s\},1) = \{s\},\\
&\delta'(\{q\},a) = \emptyset \text{ for } a \in \{0,1\},\\
&\delta'(\{s,q\},0) = \{s,q\},\ \delta'(\{s,q\},1) = \{s\}.
\end{aligned}$$

This automaton can be further optimised: The states $\emptyset$ and $\{q\}$ are never reached, hence they can be omitted from the dfa.

The next exercise shows that the exponential blow-up between the nfa and the dfa is also there when the alphabet is fixed to $\Sigma = \{0,1\}$.

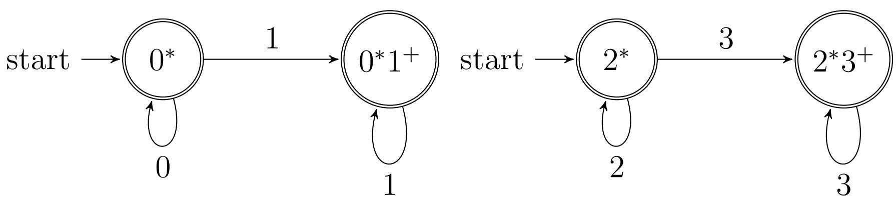
Exercise 2.43. *Consider the language* $\{0,1\}^* \cdot 0 \cdot \{0,1\}^{n-1}$*:*
(a) *Show that a dfa recognising it needs at least* 2^n *states;*
(b) *Make an nfa recognising it with at most* $n+1$ *states;*
(c) *Made a dfa recognising it with exactly* 2^n *states.*

Exercise 2.44. *Find a characterisation when a regular language* L *is recognised by an nfa only having accepting states. Examples of such languages are* $\{0,1\}^*$*,* $0^*1^*2^*$ *and* $\{1,01,001\}^* \cdot 0^*$*. The language* $\{00,11\}^*$ *is not a language of this type.*

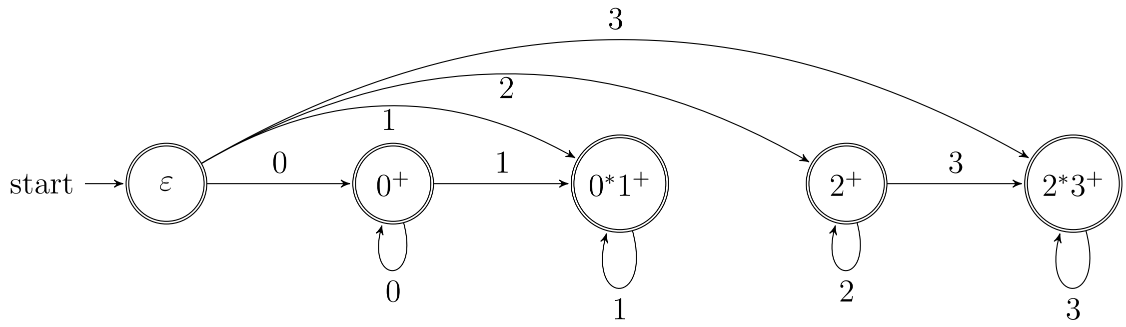
Example 2.45. One can generalise the nfa to a machine (Q,Σ,δ,I,F) where a set I of starting states replaces the single starting state s. Now such a machine accepts a string $w = a_1 a_2 \ldots a_i \in \Sigma^i$ iff there is a sequence $q_0 q_1 \ldots q_i$ of states such that

$$q_0 \in I \wedge q_i \in F \wedge \forall j < i\,[q_{i+1} \in \delta(q_i,a_i)];$$

if such a sequence does not exist then the machine rejects the input w. The following machine with three states recognises the set $0^*1^* \cup 2^*3^*$, the nodes are labelled with the regular expressions denoting the language of the words through which one can reach the corresponding node.



The corresponding nfa would need 5 states, as one needs a common start state which the nfa leaves as soon as it reads a symbol.



Exercise 2.46. *Let* $\Sigma = \{0, 1, \ldots, n-1\}$ *and* $L = \{w \in \Sigma^* :$ *some* $a \in \Sigma$ *does not occur in* $w\}$. *Show that there is a machine like in Example 2.45 with* $|Q| = n$ *which recognises* L *and that every complete dfa recognising* L *needs* 2^n *states.*

Theorem 2.47. *Every language generated by a regular grammar is also recognised by an nfa.*

Proof. If a grammar has a rule of the form $A \to w$ with w being non-empty, one can add a non-terminal B and replace the rule $A \to w$ by $A \to wB$ and $B \to \varepsilon$. Furthermore, if the grammar has a rule $A \to a_1 a_2 \ldots a_n B$ with $n > 1$ then one can introduce $n-1$ new non-terminals $C_1, C_2, \ldots, C_{n-1}$ and replace the rule by $A \to a_1 C_1$, $C_1 \to a_2 C_2$, ..., $C_{n-1} \to a_n B$. Thus if L is regular, there is a grammar (N, Σ, P, S) generating L such that all rules in P are either of the form $A \to B$ or the form $A \to aB$ or of the form $A \to \varepsilon$ where $A, B \in N$ and $a \in \Sigma$. So let such a grammar be given.

Now an nfa recognising L is given as $(N, \Sigma, \delta, S, F)$ where N and S are as in the grammar and for $A \in N, a \in \Sigma$, one defines

$$\begin{aligned} \delta(A, a) &= \{B \in N : A \Rightarrow^* aB \text{ in the grammar}\}; \\ F &= \{B \in N : B \Rightarrow^* \varepsilon\}. \end{aligned}$$

If now $w = a_1 a_2 \ldots a_n$ is a word in L then there is a derivation of the word $a_1 a_2 \ldots a_n$ of the form

$$S \Rightarrow^* a_1 A_1 \Rightarrow^* a_1 a_2 A_2 \Rightarrow^* \ldots \Rightarrow^* a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow^* a_1 a_2 \ldots a_{n-1} a_n A_n$$
$$\Rightarrow^* a_1 a_2 \ldots a_n.$$

In particular, $S \Rightarrow^* a_1 A_1$, $A_m \Rightarrow^* a_{m+1} A_{m+1}$ for all $m \in \{1, 2, \ldots, n-1\}$ and $A_n \Rightarrow^* \varepsilon$. It follows that A_n is an accepting state and $(S, A_1, A_2, \ldots, A_n)$ an accepting run of the nfa on the word $a_1 a_2 \ldots a_n$.

If now the nfa has an accepting run $(S, A_1, A_2, \ldots, A_n)$ on a word $w = a_1 a_2 \ldots a_n$ then $S \Rightarrow^* a_1 A_1$ and, for all $m \in \{1, 2, \ldots, n-1\}$, $A_m \Rightarrow^* a_{m+1} A_{m+1}$ and $A_n \Rightarrow^* \varepsilon$. It follows that $w \in L$ as witnessed by the derivation $S \Rightarrow^* a_1 A_1 \Rightarrow^* a_1 a_2 A_2 \Rightarrow^* \ldots \Rightarrow^* a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow^* a_1 a_2 \ldots a_{n-1} a_n A_n \Rightarrow^* a_1 a_2 \ldots a_n$. Thus the nfa constructed recognises the language L. ∎

Example 2.48. The language $L = 0123^*$ has a grammar with terminal alphabet $\Sigma = \{0, 1, 2, 3\}$, non-terminal alphabet $\{S, T\}$, start symbol S and rules $S \to 012|012T$, $T \to 3T|3$.

One first updates the grammar such that all rules are of the form $A \to aB$ or $A \to \varepsilon$ for $A, B \in N$ and $a \in \Sigma$. One possible updated grammar has the non-terminals $N = \{S, S', S'', S''', T, T'\}$, the start symbol S and the rules $S \to 0S'$, $S' \to 1S''$, $S'' \to 2S'''|2T$, $S''' \to \varepsilon$, $T \to 3T|3T'$, $T' \to \varepsilon$.

Now the non-deterministic finite automaton is given as $(N, \Sigma, \delta, S, \{S''', T\})$ where $\delta(S, 0) = \{S'\}$, $\delta(S', 1) = \{S''\}$, $\delta(S'', 2) = \{S''', T\}$, $\delta(T, 3) = \{T, T'\}$ and $\delta(A, a) = \emptyset$ in all other cases.

Examples for accepting runs: For $0\,1\,2$, an accepting run is $S\,(0)\,S'\,(1)\,S''\,(2)\,S'''$ and for $0\,1\,2\,3\,3\,3$, an accepting run is $S\,(0)\,S'\,(1)\,S''\,(2)\,T\,(3)\,T\,(3)\,T\,(3)\,T'$.

Exercise 2.49. *Let the regular grammar $(\{S, T\}, \{0, 1, 2\}, P, S)$ with the rules P being $S \to 01T|20S$, $T \to 01|20S|12T$. Construct a non-deterministic finite automaton recognising the language generated by this grammar.*

Exercise 2.50. *Consider the regular grammar $(\{S\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, S)$ where the rules in P are all rules of the form $S \to aaaaaS$ for some digit a and the rule $S \to \varepsilon$ and let L be the language generated by this grammar. What is the minimum number of states of a non-deterministic finite automaton recognising this language L? What is the trade-off of the nfa compared to the minimal dfa for the same language L? Prove the answers.*

Theorem 1.30 showed that a language L is generated by a regular expression iff it has a regular grammar; Theorem 2.8 showed that if L is recognised by a dfa then it L is also generated by a regular expression; Theorem 2.41 showed that if L is recognised by an nfa then L is recognised by a dfa; Theorem 2.47 showed if L is generated by a regular grammar then L is recognised by an nfa. Thus these four concepts are all equivalent.

Corollary 2.51. *A language L is regular iff it satisfies any of the following equivalent conditions:*

(a) *L is generated by a regular expression;*
(b) *L is generated by a regular grammar;*
(c) *L is recognised by a deterministic finite automaton;*
(d) *L is recognised by a non-deterministic finite automaton;*
(e) *L and its complement satisfy both the block pumping lemma;*
(f) *L satisfies Jaffe's pumping lemma;*
(g) *L has only finitely many derivatives (Theorem of Myhill and Nerode).*

It was shown above that deterministic automata can be exponentially larger than non-deterministic automata in the sense that a non-deterministic automaton with n states can only be translated into a deterministic complete automaton with 2^n states, provided that one permits multiple start states. One might therefore ask, how do the other notions relate to the size of states of automata. For the sizes of regular expressions, they depend heavily on the question of which operation one permits. Gelade and Neven [29] showed that not permitting intersection and complement in regular expressions can cause a double exponential increase in the size of the expression (measured in number of symbols to write down the expression).

Example 2.52. The language $L = \bigcup_{m<n}(\{0,1\}^m \cdot \{1\} \cdot \{0,1\}^* \cdot \{10^m\})$ can be written down in $O(n^2)$ symbols as a regular expression but the corresponding dfa has at least 2^n states: if $x = a_0 a_1 \ldots a_{n-1}$ then $10^m \in L_x$ iff $x10^m \in L$ iff $a_0 a_1 \ldots a_{n-1} 10^m \in L$ iff $a_m = 1$. Thus for $x = a_0 a_1 \ldots a_{n-1}$ and $y = b_0 b_1 \ldots b_{n-1}$, it holds that $L_x = L_y$ iff $\forall m < n\,[10^m \in L_x \Leftrightarrow 10^m \in L_y]$ iff $\forall m < n\,[a_m = b_m]$ iff $x = y$. Thus the language L has at least 2^n derivatives and therefore a dfa for L needs at least 2^n states.

One can separate regular expressions with intersections even from nfas over the unary alphabet $\{0\}$ as the following theorem shows; for this theorem, let $p_1, p_2, \ldots, p_n$ be the first n prime numbers.

Theorem 2.53. *The language $L_n = \{0^{p_1}\}^+ \cap \{0^{p_2}\}^+ \cap \ldots \cap \{0^{p_n}\}^+$ has a regular expression which can be written down with approximately $O(n^2 \log(n))$ symbols if one can use intersection. However, every nfa recognising L_n has at least 2^n states and every regular expression for L_n only using union, concatenation and Kleene star needs at least 2^n symbols.*

Proof. It is known that $p_n \leq 2 \cdot n \cdot \log(n)$ for almost all n. Each set 0^{p_m} can be written down as a regular expression consisting of two set brackets and p_m zeroes in between, if one uses Kleene star and not Kleene plus, one uses about $2p_m + 6$ symbols (two times 0^{p_m} and four set brackets and one star and one concatenation symbol, where Kleene star and plus bind stronger than concatenation, union and intersection). The n terms are then put into brackets and connected with intersection symbols what gives a total of up to $2n \cdot p_n + 3n$ symbols. So the overall number of symbols is $O(n^2 \log(n))$ in dependence of the parameter n.

The shortest word in the language must be a word of the form 0^k where each of the prime numbers $p_1, p_2, \ldots, p_n$ divides k; as all of them are distinct primes, their product is at least 2^n and the product divides k, thus $k \geq 2^n$. In an nfa, the length of the shortest accepted word is as long as the shortest path to an accepting state; in this path, each state is visited at most once and therefore the length of the shortest word is smaller than the number of states. It follows that an nfa recognising L must have at least 2^n states.

If a regular expression generating at least one word and only consisting of listed finite sets connected with union, concatenation, Kleene plus and Kleene star, then one can prove that the shortest word generated by σ is at most as long as the length of the expression. By way of contradiction, assume that σ be the length-lexicographically first regular expression such that σ generates some words, but all of these are longer than σ. Let $sw(\sigma)$ denote the shortest word generated by σ (if it exists) and if there are several, $sw(\sigma)$ is the lexicographically first of those.

- If σ is a list of words of a finite set, no word listed can be longer than σ, thus $|sw(\sigma)| \leq |\sigma|$.
- If $\sigma = (\tau \cup \rho)$ then at least one of τ, ρ is non-empty, say τ. As $|\tau| < |\sigma|$, $|sw(\tau)| \leq |\tau|$. Now $|sw(\sigma)| \leq |sw(\tau)| \leq |\tau| \leq |\sigma|$.
- If $\sigma = (\tau \cdot \rho)$ then $|\tau|, |\rho| < |\sigma|$ and $|sw(\sigma)| = |sw(\tau)| + |sw(\rho)|$, as the shortest words generated by τ and ρ concatenated give the shortest word generated by σ. It follows that $|sw(\tau)| \leq |\tau|$, $|sw(\rho)| \leq |\rho|$ and $|sw(\sigma)| = |sw(\tau)| + |sw(\rho)| \leq |\tau| + |\rho| \leq |\sigma|$.
- If $\sigma = \tau^*$ then $\varepsilon = sw(\sigma)$ and clearly $|sw(\sigma)| \leq |\sigma|$.
- If $\sigma = \tau^+$ then $sw(\sigma) = sw(\tau)$ and $|\tau| < |\sigma|$, thus $|sw(\sigma)| = |sw(\tau)| \leq |\tau| \leq |\sigma|$.

Thus in all five cases the shortest word generated by σ is at most as long as σ. It follows that any regular expression generating L and consisting only of finite sets, union, concatenation, Kleene star and Kleene plus must be at least 2^n symbols long. ∎

Exercise 2.54. *Assume that a regular expression uses lists of finite sets, Kleene star, union and concatenation and assume that this expression generates at least two words. Prove that the second-shortest word of the language generated by σ is at most as long as σ. Either prove it by structural induction or by an assumption of contradiction as in the proof before; both methods are nearly equivalent.*

Exercise 2.55. *Is Exercise 2.54 also true if one permits Kleene plus in addition to Kleene star in the regular expressions? Either provide a counter example or adjust the proof. In the case that it is not true for the bound $|\sigma|$, is it true for the bound $2|\sigma|$? Again prove that bound or provide a further counter example.*

Example 2.56: Ehrenfeucht and Zeiger's Exponential Gap [24]**.** Assume that the alphabet Σ consists of all pairs of numbers in $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$. Then a complete dfa with $n+1$ states accepts all sequences of the form $(1, a_1), (a_1, a_2), (a_2, a_3), \ldots, (a_{m-1}, a_m)$ for any numbers $a_1, a_2, \ldots, a_m$, where the automaton has the following transition-function: If it is in state a on input (a, b) then it goes to state b else it goes to state 0. The starting state is 1; the set $\{1, 2, \ldots, n\}$ is the set of accepting states and once it reaches the state 0, the automaton never leaves this state. Ehrenfeucht and Zeiger showed that any regular expression for this language needs at least 2^{n-1} symbols.

If one would permit intersection, this gap would not be there for this example, as one could write

$$
\begin{aligned}
&(\{(a,b)\cdot(b,c) : a,b,c \in \{1,2,\ldots,n\}\}^* \cdot (\varepsilon \cup \{(a,b) : a,b \in \{1,2,\ldots,n\}\})) \\
&\cap (\{(a,b) : a,b \in \{1,2,\ldots,n\}\} \cdot \{(a,b)\cdot(b,c) : a,b,c \in \{1,2,\ldots,n\}\}^* \cdot (\varepsilon \cup \\
&\{(a,b) : a,b \in \{1,2,\ldots,n\}\}))
\end{aligned}
$$

to obtain the desired expression whose size is polynomial in n.

Exercise 2.57. *Assume that an nfa of k states recognises a language L. Show that the language does then satisfy the Block Pumping Lemma (Theorem 2.9) with constant $k+1$, that is, given any words $u_0, u_1, \ldots, u_k, u_{k+1}$ such that their concatenation $u_0 u_1 \ldots u_k u_{k+1}$ is in L then there are i, j with $0 < i < j \leq k+1$ and*

$$u_0 u_1 \ldots u_{i-1} (u_i u_{i+1} \ldots u_{j-1})^* u_j u_{j+1} \ldots u_{k+1} \subseteq L.$$

Exercise 2.58. *Given numbers a, b with $a > b > 2$, Provide an example of a regular language where the Block pumping constant is b and where every nfa needs at least a states.*

3 Combining Languages

One can form new languages from old ones by combining them with basic set-theoretical operations. In most cases, the complexity in terms of the level of the Chomsky hierarchy does not change.

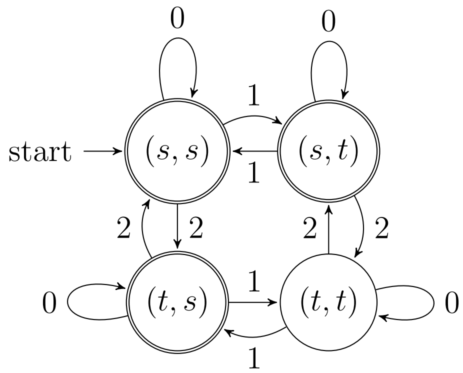
Theorem 3.1: Basic Closure Properties. *Assume that L, H are languages which are on the level CHk of the Chomsky hierarchy. Then the following languages are also on the level CHk: $L \cup H$, $L \cdot H$ and L^*.*

Description 3.2: Transforming Regular Expressions into Automata. First it is shown how to form dfas which recognise the intersection, union or difference of given sets. So let $(Q_1, \Sigma, \delta_1, s_1, F_1)$ and $(Q_2, \Sigma, \delta_2, s_2, F_2)$ be dfas which recognise L_1 and L_2, respectively.

Let $(Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (s_1, s_2), F)$ with $(\delta_1 \times \delta_2)((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ be a product automaton of the two given automata; here one can choose F such that it recognises the union or intersection or difference of the respective languages:

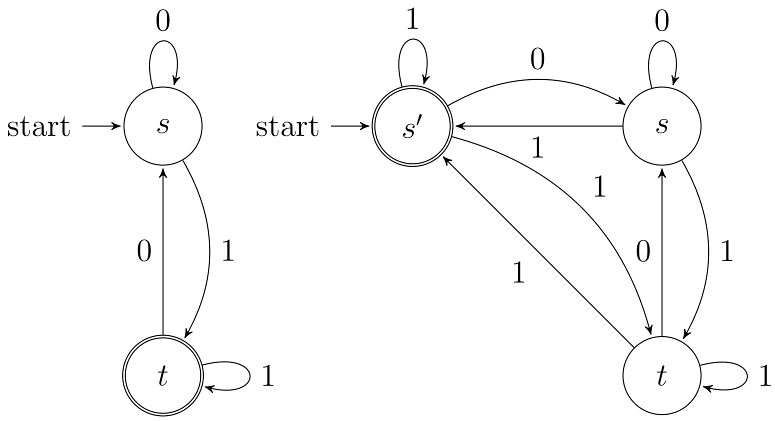
- Union: $F = F_1 \times Q_2 \cup Q_1 \times F_2$;
- Intersection: $F = F_1 \times F_2 = F_1 \times Q_2 \cap Q_1 \times F_2$;
- Difference: $F = F_1 \times (Q_2 - F_2)$;
- Symmetric Difference: $F = F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2$.

For example, let the first automaton recognise the language of words in $\{0, 1, 2\}$ with an even number of 1s and the second automaton with an even number of 2s. Both automata have the accepting and starting state s and a rejection state t; they change between s and t whenever they see 1 or 2, respectively. The product automaton is now given as follows:



The automaton given here recognises the union. For the other operations like Kleene star and concatenation, one needs to form an nfa recognising the corresponding language first and can then use Büchi's construction to transform the nfa into a dfa; as every dfa is an nfa, one can directly start with an nfa.

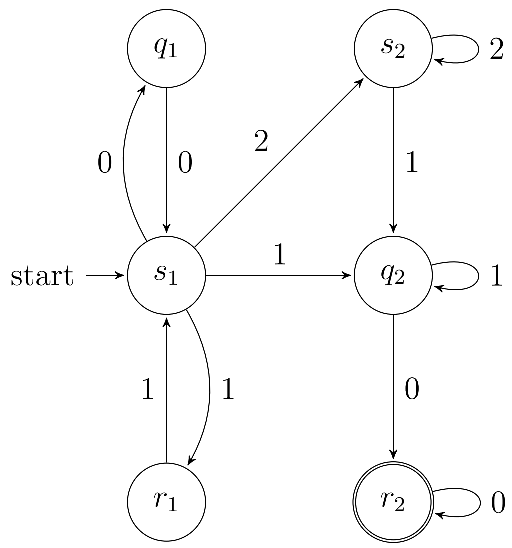
So assume $(Q, \Sigma, \delta, s, F)$ is an nfa recognising L. Now L^* is recognised by $(Q \cup \{s'\}, \Sigma, \delta', s', \{s'\})$ where $\delta' = \delta \cup \{(s', a, p) : (s, a, p) \in \delta\} \cup \{(p, a, s) : (p, a, q) \in \delta$ for some $q \in F\} \cup \{(s', a, s') : a \in L\}$. The last part of the union is to add all one-symbol words from L. This automaton has a new starting state s' which is accepting, as $\varepsilon \in L^*$. The other states in Q are kept so that the automaton can go through the states in Q in order to simulate the original automaton on some word w until it is going to process the last symbol when it then returns to s'; so it can process sequences of words in Q each time going through s'. After the last word w_n of $w_1 w_2 \ldots w_n \in L^*$, the automaton can either return to s' in order to accept the word. Here an example.



The next operation with nfas is the Concatenation. Here assume that $(Q_1, \Sigma, \delta_1, s_1, F_1)$ and $(Q_2, \Sigma, \delta_2, s_2, F_2)$ are nfas recognising L_1 and L_2 with $Q_1 \cap Q_2 = \emptyset$ and assume $\varepsilon \notin L_2$. Now $(Q_1 \cup Q_2, \Sigma, \delta, s_1, F_2)$ recognises $L_1 \cdot L_2$ where $(p, a, q) \in \delta$ whenever $(p, a, q) \in \delta_1 \cup \delta_2$ or $p \in F_1 \wedge (s_2, a, q) \in \delta_2$.

Note that if L_2 contains ε then one can consider the union of L_1 and $L_1 \cdot (L_2 - \{\varepsilon\})$.

An example is the following: $L_1 \cdot L_2$ with $L_1 = \{00, 11\}^*$ and $L_2 = 2^* 1^+ 0^+$.



Last but not least, one has to see how to build an automaton recognising a finite set, as the above only deal with the question how to get a new automaton recognising unions, differences, intersections, concatenations and Kleene star of given regular languages represented by their automata. For finite sets, one can simply consider all possible derivatives (which are easy to compute from a list of strings in the language) and then connect the corresponding states accordingly. This would indeed give the smallest dfa recognising the corresponding set.

Alternatively, one can make an automaton recognising the set $\{w\}$ and then form product automata for the unions in order to recognise sets of several strings. Here a dfa recognising $\{a_1 a_2 \ldots a_n\}$ for such a string of n symbols would have the states $q_0, q_1, \ldots, q_n$ plus r and go from q_m to q_{m+1} on input a_{m+1} and in all other cases would go to state r. Only the state q_n is accepting.

Exercise 3.3. *The above gives upper bounds on the size of the dfa for a union, intersection, difference and symmetric difference as n^2 states, provided that the original two dfas have at most n states. Give the corresponding bounds for nfas: If L and H are recognised by nfas having at most n states each, how many states does one need at most for an nfa recognising (a) the union $L \cup H$, (b) the intersection $L \cap H$, (c) the difference $L - H$ and (d) the symmetric difference $(L - H) \cup (H - L)$? Give the bounds in terms of "linear", "quadratic" and "exponential". Explain the bounds.*

Exercise 3.4. *Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Construct a (not necessarily complete) dfa recognising the language $\Sigma \cdot \{aa : a \in \Sigma\} \cap \{aaaaa : a \in \Sigma\}$. It is not needed to give a full table for the dfa, but a general schema and an explanation how it works.*

Exercise 3.5. *Make an nfa for the intersection of the following languages:* $\{0,1,2\}^* \cdot \{001\} \cdot \{0,1,2\}^* \cdot \{001\} \cdot \{0,1,2\}^*$*;* $\{001, 0001, 2\}^*$*;* $\{0,1,2\}^* \cdot \{00120001\} \cdot \{0,1,2\}^*$.

Exercise 3.6. *Make an nfa for the union* $L_0 \cup L_1 \cup L_2$ *with* $L_a = \{0,1,2\}^* \cdot \{aa\} \cdot \{0,1,2\}^* \cdot \{aa\} \cdot \{0,1,2\}^*$ *for* $a \in \{0,1,2\}$.

Exercise 3.7. *Consider two context-free grammars with terminals* Σ*, disjoint non-terminals* N_1 *and* N_2*, start symbols* $S_1 \in N_1$ *and* $S_2 \in N_2$ *and rule sets* P_1 *and* P_2 *which generate* L *and* H*, respectively. Explain how to form from these a new context-free grammar for* **(a)** $L \cup H$*,* **(b)** $L \cdot H$ *and* **(c)** L^*.

Write down the context-free grammars for $\{0^n 1^{2n} : n \in \mathbb{N}\}$ *and* $\{0^n 1^{3n} : n \in \mathbb{N}\}$ *and form the grammars for the union, concatenation and star explicitly.*

Example 3.8. The language $L = \{0^n 1^n 2^n : n \in \mathbb{N}\}$ is the intersection of the context-free languages $\{0\}^* \cdot \{1^n 2^n : n \in \mathbb{N}\}$ and $\{0^n 1^n : n \in \mathbb{N}\} \cdot \{2\}^*$. By Exercise 1.43 this language is not context-free.

Hence L is the intersection of two context-free languages which is not context-free. However, the complement of L is context-free. The following grammar generates $\{0^k 1^m 2^n : k < n\}$: the non-terminals are S, T with S being the start symbol, the terminals are $0, 1, 2$ and the rules are $S \to 0S2|S2|T2$, $T \to 1T|\varepsilon$. Now the complement of L is the union of eight context-free languages. Six languages of this type: $\{0^k 1^m 2^n : k < m\}$, $\{0^k 1^m 2^n : k > m\}$, $\{0^k 1^m 2^n : k < n\}$, $\{0^k 1^m 2^n : k > n\}$, $\{0^k 1^m 2^n : m < n\}$ and $\{0^k 1^m 2^n : m > n\}$; furthermore, the two regular languages $\{0,1,2\}^* \cdot \{10, 20, 21\} \cdot \{0,1,2\}^*$ and $\{\varepsilon\}$. So the so-constructed language is context-free while its complement L itself is not.

Although the intersection of two context-free languages might not be context-free, one can still show a weaker version of this result. This weaker version can be useful for various proofs.

Theorem 3.9. *Assume that* L *is a context-free language and* H *is a regular language. Then the intersection* $L \cap H$ *is also a context-free language.*

Proof. Assume that (N, Σ, P, S) is the context-free grammar generating L and $(Q, \Sigma, \delta, s, F)$ is the finite automaton accepting H. Furthermore, assume that every production in P is either of the form $A \to BC$ or of the form $A \to w$ for $A, B, C \in N$ and $w \in \Sigma^*$.

Now make a new grammar $(Q \times N \times Q \cup \{S\}, \Sigma, R, S)$ generating $L \cap H$ with the following rules:

- $S \to (s, S, q)$ for all $q \in F$;

- $(p, A, q) \to (p, B, r)(r, C, q)$ for all $p, q, r \in Q$ and all rules of the form $A \to BC$ in P;
- $(p, A, q) \to w$ for all $p, q \in Q$ and all rules $A \to w$ in P with $\delta(p, w) = q$.

For each $A \in N$, let $L_A = \{w \in \Sigma^* : S \Rightarrow^* w\}$. For each $p, q \in Q$, let $H_{p,q} = \{w \in \Sigma^* : \delta(p, w) = q\}$. Now one shows that (p, A, q) generates w in the new grammar iff $w \in L_A \cap H_{p,q}$.

First one shows by induction over every derivation-length that a symbol (p, A, q) can only generate a word w iff $\delta(p, w) = q$ and $w \in L_A$. If the derivation-length is 1 then there is a production $(p, A, q) \to w$ in the grammar. It follows from the definition that $\delta(p, w) = q$ and $A \to w$ is a rule in P, thus $w \in L_A$. If the derivation-length is larger than 1, then one uses the induction hypothesis that the statement is already shown for all shorter derivations and now looks at the first rule applied in the derivation. It is of the form $(p, A, q) \to (q, B, r)(r, C, q)$ for some $B, C \in N$ and $r \in Q$. Furthermore, there is a splitting of w into uv such that (q, B, r) generates u and (r, C, q) generates v. By induction hypothesis and the construction of the grammar, $u \in L_B$, $v \in L_C$, $\delta(p, u) = r$, $\delta(r, v) = q$ and $A \to BC$ is a rule in P. It follows that $A \Rightarrow BC \Rightarrow^* uv$ in the grammar for L and $w \in L_A$. Furthermore, $\delta(p, uv) = \delta(r, v) = q$, hence $w \in H_{p,q}$. This completes the proof of this part.

Second one shows that the converse holds, now by induction over the length of derivations in the grammar for L. Assume that $w \in L_A$ and $w \in H_{p,q}$. If the derivation has length 1 then $A \to w$ is a rule the grammar for L. As $\delta(p, w) = q$, it follows that $(p, A, q) \to w$ is a rule in the new grammar. If the derivation has length $n > 1$ and the proof has already been done for all derivations shorter than n, then the first rule applied to show that $w \in L_A$ must be a rule of the form $A \to BC$. There are $u \in L_B$ and $v \in L_C$ with $w = uv$. Let $r = \delta(p, u)$. It follows from the definition of δ that $q = \delta(r, v)$. Hence, by induction hypothesis, (p, B, r) generates u and (r, C, q) generates v. Furthermore, the rule $(p, A, q) \to (p, B, r)(r, C, q)$ is in the new grammar, hence (p, A, q) generates $w = uv$.

Now one has for each $p, q \in Q$, $A \in N$ and $w \in \Sigma^*$ that (p, A, q) generates w iff $w \in L_A \cap H_{p,q}$. Furthermore, in the new grammar, S generates a string w iff there is a $q \in F$ with (s, S, q) generating w iff $w \in L_S$ and $\delta(s, w) \in F$ iff $w \in L_S$ and there is a $q \in F$ with $w \in H_{s,q}$ iff $w \in L \cap H$. This completes the proof. ∎

Exercise 3.10. *Recall that the language L of all words which contain as many 0s as 1s is context-free; a grammar for it is $(\{S\}, \{0, 1\}, \{S \to SS|\varepsilon|0S1|1S0\}, S)$. Construct a context-free grammar for $L \cap (001^+)^*$.*

Exercise 3.11. *Let again L be the language of all words which contain as many 0s as 1s. Construct a context-free grammar for $L \cap 0^*1^*0^*1^*$.*

Theorem 3.12. *The concatenation of two context-sensitive languages is context-sensitive.*

Proof. Let L_1 and L_2 be context-sensitive languages not containing ε and consider context-sensitive grammars (N_1, Σ, P_1, S_1) and (N_2, Σ, P_2, S_2) generating L_1 and L_2, respectively, where $N_1 \cap N_2 = \emptyset$ and where each rule $l \to r$ satisfies $|l| \leq |r|$ and $l \in N_e^+$ for the respective $e \in \{1, 2\}$. Let $S \notin N_1 \cup N_2 \cup \Sigma$. Now the automaton

$$(N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

generates $L_1 \cdot L_2$: If $v \in L_1$ and $w \in L_2$ then $S \Rightarrow S_1 S_2 \Rightarrow^* vS_2 \Rightarrow^* vw$. Furthermore, the first rule has to be $S \Rightarrow S_1 S_2$ and from then onwards, each rule has on the left side either $l \in N_1^*$ so that it applies to the part generated from S_1 or it has in the left side $l \in N_2^*$ so that l is in the part of the word generated from S_2. Hence every intermediate word z in the derivation is of the form $xy = z$ with $S_1 \Rightarrow^* x$ and $S_2 \Rightarrow^* y$.

In the case that one wants to form $(L_1 \cup \{\varepsilon\}) \cdot L_2$, one has to add the rule $S \to S_2$, for $L_1 \cdot (L_2 \cup \{\varepsilon\})$, one has to add the rule $S \to S_1$ and for $(L_1 \cup \{\varepsilon\}) \cdot (L_2 \cup \{\varepsilon\})$, one has to add the rules $S \to S_1 | S_2 | \varepsilon$ to the grammar. ∎

As an example consider the following context-sensitive grammars generating two sets L_1 and L_2 not containing the empty string ε, the second grammar could also be replaced by a context-free grammar but is here only chosen to be context-sensitive:

- $(\{S_1, T_1, U_1, V_1\}, \{0, 1, 2, 3, 4\}, P_1, S_1)$ with P_1 containing the rules $S_1 \to T_1 U_1 V_1 S_1 \mid T_1 U_1 V_1$, $T_1 U_1 \to U_1 T_1$, $T_1 V_1 \to V_1 T_1$, $U_1 T_1 \to T_1 U_1$, $U_1 V_1 \to V_1 U_1$, $V_1 T_1 \to T_1 V_1$, $V_1 U_1 \to U_1 V_1$, $T_1 \to 0$, $V_1 \to 1$, $U_1 \to 2$ generating all words with the same nonzero number of 0s, 1s and 2s;

- $(\{S_2, T_2, U_2\}, \{0, 1, 2, 3, 4\}, P_2, S_2)$ with P_2 containing the rules $S_2 \to U_2 T_2 S_2 \mid U_2 T_2$, $U_2 T_2 \to T_2 U_2$, $T_2 U_2 \to U_2 T_2$, $U_2 \to 3$, $T_2 \to 4$ generating all words with the same nonzero number of 3s and 4s.

The grammar $(\{S, S_1, T_1, U_1, V_1, S_2, T_2, U_2\}, \{0, 1, 2, 3, 4\}, P, S)$ with P containing $S \to S_1 S_2$, $S_1 \to T_1 U_1 V_1 S_1 | T_1 U_1 V_1$, $T_1 U_1 \to U_1 T_1$, $T_1 V_1 \to V_1 T_1$, $U_1 T_1 \to T_1 U_1$, $U_1 V_1 \to V_1 U_1$, $V_1 T_1 \to T_1 V_1$, $V_1 U_1 \to U_1 V_1$, $T_1 \to 0$, $V_1 \to 1$, $U_1 \to 2$, $S_2 \to U_2 T_2 S_2 | U_2 T_2$, $U_2 T_2 \to T_2 U_2$, $T_2 U_2 \to U_2 T_2$, $U_2 \to 3$, $T_2 \to 4$ generates all words with consisting of n 0s, 1s and 2s in any order followed by m 3s and 4s in any order with $n, m > 0$. For example, 01120234434334 is a word in this language. The grammar is context-sensitive in the sense that $|l| \leq |r|$ for all rules $l \to r$ in P.

Theorem 3.13. *If L is context-sensitive so is L^*.*

Proof. Assume that (N_1, Σ, P_1, S_1) and (N_2, Σ, P_2, S_2) are two context-sensitive grammars for L with $N_1 \cap N_2 = \emptyset$ and all rules $l \to r$ satisfying $|l| \leq |r|$ and $l \in N_1^+$ or $l \in N_2^+$, respectively. Let S, S' be symbols not in $N_1 \cup N_2 \cup \Sigma$.

The new grammar is of the form $(N_1 \cup N_2 \cup \{S, S'\}, \Sigma, P, S)$ where P contains the rules $S \to S'|\varepsilon$ and $S' \to S_1S_2S' \mid S_1S_2 \mid S_1$ plus all rules in $P_1 \cup P_2$.

The overall idea is the following: if $w_1, w_2, \ldots, w_{2n}$ are non-empty words in L, then one generates $w_1w_2 \ldots w_{2n}$ by first generating the string $(S_1S_2)^n$ using the rule $S \to S'$, $n-1$ times the rule $S' \to S_1S_2S'$ and one time the rule $S' \to S_1S_2$. Afterwords one derives inductively S_1 to w_1, then the next S_2 to w_2, then the next S_1 to w_3, $\ldots$, until one has achieved that all S_1 and S_2 are transformed into the corresponding w_m.

The alternations between S_1 and S_2 are there to prevent that one can non-terminals generated for a word w_k and for the next word w_{k+1} mix in order to derive something what should not be derived. So only words in L^* can be derived. ∎

Exercise 3.14. *Recall that the language $L = \{0^n1^n2^n : n \in \mathbb{N}\}$ is context-sensitive. Construct a context-sensitive grammar for L^*.*

The next result gives an explicit way to construct the intersection of two context-sensitive languages; for the construction, only an example is given, one can generalise this construction to get the full result.

Example 3.15. *Let $Eq_{a,b}$ be the language of all non-empty words w over Σ such that w contains as many a as b where $a, b \in \Sigma$. Let $\Sigma = \{0, 1, 2\}$ and $L = Eq_{0,1} \cap Eq_{0,2}$. The language L is context-sensitive.*

Proof. First one makes a grammar for $Eq_{a,b}$ where c stands for any symbol in $\Sigma - \{a, b\}$. The grammar has the form

$$(\{S\}, \Sigma, \{S \to SS|aSb|bSa|ab|ba|c\}, S)$$

and one now makes a new grammar for the intersection as follows: The idea is to produce two-componented characters where the upper component belongs to a derivation of $Eq_{0,1}$ and the lower belongs to a derivation of $Eq_{0,2}$. Furthermore, there will in both components be a space symbol, #, which can be produced on the right side of the start symbol in the beginning and later be moved from the right to the left. Rules which apply only to the upper or lower component do not change the length, they just eat up some spaces if needed. Then the derivation is done on the upper and lower part independently. In the case that the outcome is on the upper and the lower component the same, the whole word is then transformed into the corresponding symbols from

Σ.

The non-terminals of the new grammar are all of the form $\binom{A}{B}$ where $A, B \in \{S, \#, 0, 1, 2\}$. In general, each non-terminal represents a pair of a symbols which can occur in the upper and lower derivation; pairs are by definition different from terminals in $\Sigma = \{0, 1, 2\}$. The start symbol is $\binom{S}{S}$. The following rules are there:

1. The rule $\binom{S}{S} \to \binom{S}{S}\binom{\#}{\#}$. This rule permits to produce space right of the start symbol which is later used independently in the upper or lower component. For each symbols A, B, C in $\{S, \#, 0, 1, 2\}$ one introduces the rules $\binom{A}{B}\binom{\#}{C} \to \binom{\#}{B}\binom{A}{C}$ and $\binom{A}{C}\binom{B}{\#} \to \binom{A}{\#}\binom{B}{C}$ which enable to bring, independently of each other, the spaces in the upper and lower component from the right to the left.

2. The rules of $Eq_{0,1}$ will be implemented in the upper component. If a rule of the form $l \to r$ has that $|l| + k = |r|$ then one replaces it by $l\#^k \to r$. Furthermore, the rules have now to reflect the lower component as well, so there are entries which remain unchanged but have to be mentioned. Therefore one adds for each choice of $A, B, C \in \{S, \#, 0, 1, 2\}$ the following rules into the set of rules of the grammar:
$\binom{S}{A}\binom{\#}{B} \to \binom{S}{A}\binom{S}{B}$, $\binom{S}{A}\binom{\#}{B}\binom{\#}{C} \to \binom{0}{A}\binom{S}{B}\binom{1}{C} \mid \binom{1}{A}\binom{S}{B}\binom{0}{C}$,
$\binom{S}{A}\binom{\#}{B} \to \binom{0}{A}\binom{1}{B} \mid \binom{1}{A}\binom{0}{B}$, $\binom{S}{A} \to \binom{2}{A}$;

3. The rules of $Eq_{0,2}$ are implemented in the lower component and one takes again for all $A, B, C \in \{S, \#, 0, 1, 2\}$ the following rules into the grammar:
$\binom{A}{S}\binom{B}{\#} \to \binom{A}{S}\binom{B}{S}$, $\binom{A}{S}\binom{B}{\#}\binom{C}{\#} \to \binom{A}{0}\binom{B}{S}\binom{C}{2} \mid \binom{A}{2}\binom{B}{S}\binom{C}{0}$,
$\binom{A}{S}\binom{B}{\#} \to \binom{A}{0}\binom{B}{2} \mid \binom{A}{2}\binom{B}{0}$, $\binom{A}{S} \to \binom{A}{1}$;

4. To finalise, one has the rule $\binom{a}{a} \to a$ for each $a \in \Sigma$, that is, the rules $\binom{0}{0} \to 0$, $\binom{1}{1} \to 1$, $\binom{2}{2} \to 2$ in order to transform non-terminals consisting of matching placeholders into the corresponding terminals. Non-matching placeholders and spaces cannot be finalised, if they remain in the word, the derivation cannot terminate.

To sum up, a word $w \in \Sigma^*$ can only be derived iff w is derived independently in the upper and the lower component of the string of non-terminals according to the rules of $Eq_{0,1}$ and $Eq_{0,2}$. The resulting string of pairs of matching entries from Σ is then transformed into the word w.

The following derivation of the word 011022 illustrates the way the word is generated: in the first step, enough space is produced; in the second step, the upper component is derived; in the third step, the lower component is derived; in the fourth step, the terminals are generated from the placeholders.

1. $\binom{S}{S} \Rightarrow \binom{S}{S}\binom{\#}{\#} \Rightarrow \binom{S}{S}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow \binom{S}{S}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow \binom{S}{S}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow$ $\binom{S}{S}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow$
2. $\binom{S}{S}\binom{S}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow \binom{S}{S}\binom{2}{\#}\binom{2}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#} \Rightarrow^* \binom{S}{S}\binom{\#}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow$ $\binom{S}{S}\binom{S}{\#}\binom{\#}{\#}\binom{\#}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow \binom{S}{S}\binom{\#}{\#}\binom{S}{\#}\binom{\#}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow \binom{0}{S}\binom{1}{\#}\binom{S}{\#}\binom{\#}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow$ $\binom{0}{S}\binom{1}{\#}\binom{1}{\#}\binom{0}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow$
3. $\binom{0}{0}\binom{1}{S}\binom{1}{2}\binom{0}{\#}\binom{2}{\#}\binom{2}{\#} \Rightarrow^* \binom{0}{0}\binom{1}{S}\binom{1}{\#}\binom{0}{\#}\binom{2}{\#}\binom{2}{2} \Rightarrow \binom{0}{0}\binom{1}{S}\binom{1}{S}\binom{0}{\#}\binom{2}{\#}\binom{2}{2} \Rightarrow$ $\binom{0}{0}\binom{1}{1}\binom{1}{S}\binom{0}{\#}\binom{2}{\#}\binom{2}{2} \Rightarrow \binom{0}{0}\binom{1}{1}\binom{1}{S}\binom{0}{S}\binom{2}{\#}\binom{2}{2} \Rightarrow \binom{0}{0}\binom{1}{1}\binom{1}{1}\binom{0}{S}\binom{2}{\#}\binom{2}{2} \Rightarrow$ $\binom{0}{0}\binom{1}{1}\binom{1}{1}\binom{0}{0}\binom{2}{2}\binom{2}{2} \Rightarrow$
4. $0\binom{1}{1}\binom{1}{1}\binom{0}{0}\binom{2}{2}\binom{2}{2} \Rightarrow 01\binom{1}{1}\binom{0}{0}\binom{2}{2}\binom{2}{2} \Rightarrow 011\binom{0}{0}\binom{2}{2}\binom{2}{2} \Rightarrow 0110\binom{2}{2}\binom{2}{2} \Rightarrow$ $01102\binom{2}{2} \Rightarrow 011022.$

In this derivation, each step is shown except that several moves of characters in components over spaces are put together to one move. ∎

The proof of the following theorem is omitted, it is mainly obtained by formalising the methods of the previous example for arbitrary grammars instead of the two concrete ones. Note that here instead of the original context-sensitiveness-condition the modified condition is used that each rule $l \to r$ satisfies $|l| \leq |r|$. Indeed, in the grammar produced, all rules satisfy $|l| = |r|$ except for the initial rule which produces pairs of spaces from the starting symbol. One can also cover the case that both languages contain the empty word with a small additional reasoning and modification of the resulting grammar.

Theorem 3.16. *The intersection of context-sensitive languages is context-sensitive.*

Exercise 3.17. *Consider the language* $L = \{00\} \cdot \{0,1,2,3\}^* \cup \{1,2,3\} \cdot \{0,1,2,3\}^* \cup \{0,1,2,3\}^* \cdot \{02,03,13,10,20,30,21,31,32\} \cdot \{0,1,2,3\}^* \cup \{\varepsilon\} \cup \{01^n2^n3^n : n \in \mathbb{N}\}$. *Which of the pumping conditions from Theorems 1.35 (a) and 1.35 (b), Corollary 1.36 and Theorem 2.9 does the language satisfy? Determine its exact position in the Chomsky hierarchy.*

Exercise 3.18. *Let* x^{mi} *be the mirror image of* x*, so* $(01001)^{mi} = 10010$*. Furthermore, let* $L^{mi} = \{x^{mi} : x \in L\}$*. Show the following two statements:*
(a) *If an nfa with* n *states recognises* L *then there is also an nfa with up to* $n+1$ *states recognising* L^{mi}*.*
(b) *Find the smallest nfas which recognise* $L = 0^*(1^* \cup 2^*)$ *as well as* L^{mi}*.*

Description 3.19: Palindromes. The members of the language $\{x \in \Sigma^* : x = x^{mi}\}$ are called palindromes. A palindrome is a word or phrase which looks the same from both directions.

An example is the German name "OTTO"; furthermore, when ignoring spaces and punctuation marks, a famous palindrome is the phrase "A man, a plan, a canal: Panama." originating from the time when the canal in Panama was built.

The grammar with the rules $S \to aSa|aa|a|\varepsilon$ with a ranging over all members of Σ generates all palindromes; so for $\Sigma = \{0,1,2\}$ the rules of the grammar would be $S \to 0S0\,|\,1S1\,|\,2S2\,|\,00\,|\,11\,|\,22\,|\,0\,|\,1\,|\,2\,|\,\varepsilon$.

The set of palindromes is not regular. This can easily be seen by the pumping lemma, as otherwise $L \cap 0^*10^* = \{0^n10^n : n \in \mathbb{N}\}$ would have to be regular. However, this is not the case, as there is a constant k such that one can pump the word 0^k10^k by omitting some of the first k characters; the resulting word 0^h10^k with $h < k$ is not in L as it is not a palindrome. Hence L does not satisfy the pumping lemma when the word has to be pumped among the first k characters.

Exercise 3.20. *Let $w \in \{0,1,2,3,4,5,6,7,8,9\}^*$ be a palindrome of even length and n be its decimal value. Prove that n is a multiple of 11. Note that it is essential that the length is even, as for odd length there are counter examples (like 111 and 202).*

Exercise 3.21. *Given a context-free grammar for a language L, is there also one for $L \cap L^{mi}$? If so, explain how to construct the grammar; if not, provide a counter example where L is context-free but $L \cap L^{mi}$ is not.*

Exercise 3.22. *Is the following statement true or false? Prove the answer: Given a language L, the language $L \cap L^{mi}$ equals to $\{w \in L : w$ is a palindrome$\}$.*

Exercise 3.23. *Let $L = \{w \in \{0,1,2\}^* : w = w^{mi}\}$ and consider $H = L \cap \{012, 210, 00, 11, 22\}^* \cap (\{0,1\}^* \cdot \{1,2\}^* \cdot \{0,1\}^*)$. This is the intersection of a context-free and regular language and thus context-free. Construct a context-free grammar for H.*

Definition 3.24. *Let $PUMP_{sw}$, $PUMP_{st}$ and $PUMP_{bl}$ be the classes of languages which can be pumped somewhere as formalised in Corollary 1.36, pumped at the start as formalised in Theorem 1.35 (a) and pumped according to a splitting in blocks as described in the Block Pumping Lemma (Theorem 2.9), respectively.*

The proof of Theorem 1.35 (a) showed that when L and H are in PUMP_{st} then so are $L \cup H$, $L \cdot H$, L^* and L^+.

Proposition 3.25. *The classes $PUMP_{sw}$ and $PUMP_{st}$ are closed under union, concatenation, Kleene star and Kleene plus.*

The next example establishes that these two classes are not closed under intersection, even not under intersection with regular languages.

Example 3.26. Consider $L = \{0^h 1^k 2^m 3^n : h = 0 \text{ or } k = m = n\}$ and consider $H = \{00\} \cdot \{1\}^* \cdot \{2\}^* \cdot \{3\}^*$. The language L is in PUMP$_{sw}$ and PUMP$_{st}$ and so is the language H as the latter is regular; however, the intersection $L \cap H = \{0^2 1^n 2^n 3^n : n \geq 0\}$ does not satisfy any pumping lemma. Furthermore, L^{mi} is not in PUMP$_{st}$.

Proposition 3.27. *If L is in PUMP$_{sw}$ so is L^{mi}; if L is in PUMP$_{bl}$ so is L^{mi}.*

Exercise 3.28. *Show that PUMP$_{bl}$ is closed under union and concatenation. Furthermore, show that the language $L = \{v3w4 : v, w \in \{0,1,2\}^*$ and if v, w are both square-free then $|v| \neq |w|$ or $v = w\}$ is in PUMP$_{bl}$ while L^+ and L^* are not.*

Theorem 3.29: Chak, Freivalds, Stephan and Tan [12]. *If L, H are in PUMP$_{bl}$ so is $L \cap H$.*

Proof. The proof uses Ramsey's Theorem of Pairs. Recall that when one splits a word x into blocks $u_0, u_1, \ldots, u_p$ then the borders between the blocks are called breakpoints; furthermore, $u_1, \ldots, u_{p-1}$ should not be empty (otherwise one could pump the empty block).

Ramsey's Theorem of pairs says now that for every number c there is a number $c' > c$ such that given a word x with a set I of c' breakpoints, if one colours each pair (i, j) of breakpoints (pairs have always i strictly before j) in one of the colours "white" and "red", then one can select a subset $J \subseteq I$ of c breakpoints and a colour $q \in \{\text{white,red}\}$ such that each pair of the breakpoints in J has the colour q.

Now the idea is the following: Let c be a common upper bound of the two block pumping constants for L and H, this c is then also a valid block pumping constant. Then choose c' according to Ramsey's Theorem of Pairs and consider a word $x \in L \cap H$ split into $c' + 1$ parts by a set I of c' breakpoints. Now for each pair of breakpoints $i, j \in I$ splitting x into u, v, w, let the colour "white" denote that $u \cdot v^* \cdot w \subseteq L$ and "red" that this is not the case. By Ramsey's Theorem of Pairs there is a subset $J \subseteq I$ of c breakpoints which split x and a colour q such that each pair of breakpoints in J has colour q. As J consists of c breakpoints, there must be a pair (i, j) of breakpoints in J splitting x into $u \cdot v \cdot w$ with $u \cdot v^* \cdot w \subseteq L$, thus the colour q is white and therefore every pair of breakpoints in J has this property.

Now, as c is also the block pumping constant for H, there is a pair (i, j) of breakpoints in J which splits the word into u, v, w such that $u \cdot v^* \cdot w \subseteq H$. As seen before, $u \cdot v^* \cdot w \subseteq L$ and thus $u \cdot v^* \cdot w \subseteq L \cap H$. Thus $L \cap H$ is satisfies the Block Pumping Lemma with constant c' and $L \cap H$ is in PUMP$_{bl}$. ∎

Selftest 3.30. *Consider the language L of all words of the form $uvvvw$ with $u, v, w \in \{0, 1, 2\}^*$ and $0 < |v| < 1000000$. Is this language (a) regular or (b) context-free and not regular or (c) context-sensitive and not context-free? Choose the right option and explain the answer.*

Selftest 3.31. *Consider the language L from Selftest 3.30. Does this language satisfy the traditional pumping lemma (Theorem 1.35 (a)) for regular languages? If so, what is the optimal constant?*

Selftest 3.32. *Construct a deterministic finite automaton which checks whether a decimal number is neither divisible by 3 nor by 5. This automaton does not need to exclude numbers with leading zeroes. Make the automaton as small as possible.*

Selftest 3.33. *Construct by structural induction a function F which translates regular expressions for subsets of $\{0, 1, 2, \ldots, 9\}^*$ into regular expressions for subsets of $\{0\}^*$ such that the language of $F(\sigma)$ contains the word 0^n iff the language of σ contains some word of length n.*

Selftest 3.34. *Assume that an non-deterministic finite automaton has 1000 states and accepts some word. How long is, in the worst case, the shortest word accepted by the automaton?*

Selftest 3.35. *What is the best block pumping constant for the language L of all words which contain at least three zeroes and at most three ones?*

Selftest 3.36. *Construct a context-free grammar which recognises all the words $w \in \{00, 01, 10, 11\}^*$ which are not of the form vv for any $v \in \{0, 1\}^*$.*

Selftest 3.37. *Construct a constext-sensitive grammar which accepts a word iff it has the same amount of 0, 1 and 2.*

Solution for Selftest 3.30. The right answer is (a). One can write the language as an extremely long regular expression of the form $\{0,1,2\}^* \cdot v_0 v_0 \cdot \{0,1,2\}^* \cup \{0,1,2\}^* \cdot v_1 v_1 \cdot \{0,1,2\}^* \cup \ldots \cup \{0,1,2\}^* \cdot v_n v_n \cdot \{0,1,2\}^*$. Here $v_0, v_1, \ldots, v_n$ is a list of all ternary strings from length 1 to 999999 and there are $(3^{1000000} - 3)/2$ of them. Although this expression can be optimised a bit, there is no really small one for the language which one can write down explicitly.

Solution for Selftest 3.31. For the language L from Selftest 3.30, the optimal constant for the traditional pumping lemma is 3:

If a word contains 3 symbols and is in L then it is of the form abb or aab; in the first case a^*bb and in the second case aab^* are subsets of the language. So now assume that a word in the language L is given and it has at least four symbols.

(a) The word is of the form $abbw$ or $aabw$ for any $w \in \{0,1,2\}^*$ and $a, b \in \{0,1,2\}$. This case matches back to the three-letter case and a^*bbw or aab^*w are then languages resulting by pumping within the first three symbols which prove that the language satisfies the pumping lemma with this constant.

(b) The word is of the form $auvvw$ for some $u, v, w \in \{0,1,2\}^*$ with $0 < |v| < 1000000$. In this case, a^*uvvw is a subset of the language and the pumping constant is met.

(c) The word is of the form $abaw$ for some $w \in \{0,1,2\}^*$. Then $ab^*aw \subseteq L$, as when one omits the b then it starts with aa and when one repeats the b it has the subword bb. So also in this case the pumping constant is met.

(d) The word is of the form $abcbw$, then $abc^*bw \subseteq L$ and the pumping is in the third symbol and the pumping constant is met.

(e) The word is of the form $abcaw$ for some $w \in \{0,1,2\}^*$ then $a(bc)^*aw \subseteq L$. If one omits the pump bc then the resulting start starts with aa and if one repeats the pump then the resulting word has the subword $bcbc$.

One can easily verify that this case distinction is exhaustive (with a, b, c ranging over $\{0,1,2\}$). Thus in each case where the word is in L, one can find a pumping which involves only positions within its first three symbols.

Solution for Selftest 3.32. The automaton has five states named s_0, s_1, s_2, q_1, q_2. The start state is s_0 and the set of accepting states is $\{q_1, q_2\}$. The goal is that after processing a number w with remainder a by 3, if this number is a multiple of 3 or of 5 then the automaton is in the state s_a else it is in the state q_a. Let c denote the remainder of $(a + b)$ at division by 3, where b is the decimal digit on the input. Now one can define the transition function δ as follows: If $b \in \{0,5\}$ or $c = 0$ then $\delta(s_a, b) = \delta(q_a, b) = s_c$ else $\delta(s_a, b) = \delta(q_a, b) = q_c$. Here the entry for $\delta(q_a, b)$ has to be ignored if $a = 0$.

The explanation behind this automaton is that the last digit reveals whether the number is a multiple of five and that the running sum modulo three reveals whether the number is a multiple of 3. Thus there are two sets of states, s_0, s_1, s_2 which store the remainder by 3 in the case that the number is a multiple of five and q_0, q_1, q_2 which store the remainder by 3 in the case that the number is not a multiple of five. By assumption only the states q_1, q_2 are accepting. Above rules state how to update the states. As s_0, q_0 are both rejecting and as they have in both cases the same successors, one can fusionate these two states and represent them by s_0 only, thus only five states are needed. Note that s_1 and q_1 differ as one is accepting and one is rejecting, similarly s_2 and q_2. Furthermore, given $a \in \{0, 1, 2\}$, the digit $b = 3 - a$ transfers from s_c, q_c into a rejecting state iff $c = a$, hence the states s_0, s_1, s_2 are all different and similarly q_1, q_2. So one cannot get a smaller finite automaton for this task.

Solution for Selftest 3.33. In the following definition it is permitted that elements of sets are listed multiply in a set encoded into a regular expression. So one defines F as follows:

$$\begin{array}{rcl} F(\emptyset) & = & \emptyset; \\ F(\{w_1, \ldots, w_n\}) & = & \{0^{|w_1|}, 0^{|w_2|}, \ldots, 0^{|w_n|}\}; \\ F((\sigma \cup \tau)) & = & (F(\sigma) \cup F(\tau)); \\ F((\sigma \cdot \tau)) & = & (F(\sigma) \cdot F(\tau)); \\ F((\sigma)^*) & = & (F(\sigma))^*. \end{array}$$

Here bracketing conventions from the left side are preserved. One could also define everything without a structural induction by saying that in a given regular expression, one replaces every occurrence of a digit (that is, 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9) by 0.

Solution for Selftest 3.34. First one considers the non-deterministic finite automaton with states $\{s_0, s_1, \ldots, s_{999}\}$ such that the automaton goes, on any symbol, from s_e to s_{e+1} in the case that $e < 999$ and from s_{999} to s_0. Furthermore, s_{999} is the only accepting state. This nfa is actually a dfa and it is easy to see that all accepted words have the length $k \cdot 1000 + 999$, so the shortest accepted word has length 999.

Second assume that an nfa with 1000 states is given and that x is a shortest accepted word. There is a shortest run on this nfa for x. If there are two different prefixes u, uv of x such that the nfa at this run is in the same state and if $x = uvw$ then the nfa also accepts the word uw, hence x would not be the shortest word. Thus, all the states of the nfa including the first one before starting the word and the last one after completely processing the word are different; thus the number of symbols in x is at least one below the number of states of the nfa and therefore $|x| \leq 999$.

Solution for Selftest 3.35. First one shows that the constant must be larger than 7. So assume it would be 7 and consider the word 000111 which is in the language.

One can now choose $u_0 = \varepsilon$, $u_1 = 0$, $u_2 = 0$, $u_3 = 0$, $u_4 = 1$, $u_5 = 1$, $u_6 = 1$, $u_7 = \varepsilon$ and $000111 = u_0u_1u_2u_3u_4u_5u_6u_7$. The next is to show that there are no i, j with $0 < i < j \leq 7$ such that $u_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^* u_j \ldots u_7$ is a subset of the language. If $u_i \ldots u_{j-1}$ contains at least one 1 then $_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^2 u_j \ldots u_7$ would not be in the language as it contains too many ones. If $u_i \ldots u_{j-1}$ contains at least one 0 then $_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^0 u_j \ldots u_7$ would not be in the language as it contains not enough zeroes. Thus the block pumping constant cannot be 7.

Second one shows that the constant can be chosen to be 8. Given any word $u_0u_1u_2u_3u_4u_5u_6u_7u_8$ of the language, only three of the blocks u_k can contain a 1. Furthermore, the blocks $u_1, u_2, u_3, u_4, u_5, u_6, u_7$ must be non-empty, as otherwise one could pump the empty block. So at least four of these blocks contain a 0. Thus one can pump any single block which does not contain a one, as there remain three further blocks containing at least one 0 and the number of ones is not changed; it follows that the block pumping constant is 8.

Solution for Selftest 3.36. This is a famous language. The grammar is made such that there are two different symbols a, b such that between these are as many symbols as either before a or behind b; as there are only two symbols, one can choose $a = 0$ and $b = 1$. The grammar is the following: $(\{S, T, U\}, \{0, 1\}, P, S)$ with P containing the rules $S \to TU|UT$, $U \to 0U0|0U1|1U0|1U1|0$ and $T \to 0T0|0T1|1T0|1T1|1$. Now U produces 0 and T produces 1 and before terminalising, these symbols produce the same number of symbols before and after them. So for each word generated there are $n, m \in \mathbb{N}$ such that the word is in $\{0,1\}^n \cdot \{0\} \cdot \{0,1\}^{n+m} \cdot \{1\} \cdot \{0,1\}^m$ or in $\{0,1\}^n \cdot \{1\} \cdot \{0,1\}^{n+m} \cdot \{0\} \cdot \{0,1\}^m$; in both cases, the symbols at positions n and $(n + m + 1) + n$ are different where the word itself has length $2(n + m + 1)$, thus the word cannot be of the form vv. If a word $w = uv$ with $|u| = |v| = k$ and u, v differ at position n then one lets $m = k - n - 1$ and shows that the grammar can generate uv with parameters n, m as given.

Solution for Selftest 3.37. The grammar contains the non-terminals S, T and the rules $S \to \varepsilon, 01T$ and $T \to T012|2$ and, for all $a, b \in \{0, 1, 2\}$, the rules $Ta \to aT$, $Ta \to aT$, $Tab \to bTa$, $bTa \to Tab$. The start-symbol is S and the terminal alphabet is $\{0, 1, 2\}$.

4 Homomorphisms

A homomorphism is a mapping which replaces each character by a word. In general, they can be defined as follows.

Definition 4.1: Homomorphism. *A homomorphism is a mapping h from words to words satisfying $h(xy) = h(x) \cdot h(y)$ for all words x, y.*

Proposition 4.2. When defined on words over an alphabet Σ, the values $h(a)$ for the $a \in \Sigma$ define the image $h(w)$ of every word w.

Proof. As $h(\varepsilon) = h(\varepsilon \cdot \varepsilon) = h(\varepsilon) \cdot h(\varepsilon)$, the word $h(\varepsilon)$ must also be the empty word ε. Now one can define inductively, for words of length $n = 0, 1, 2, \ldots$ the value of h: For words of length 0, $h(w) = \varepsilon$. When h is defined for words of length n, then every word $w \in \Sigma^{n+1}$ is of the form va for $v \in \Sigma^n$ and $a \in \Sigma$, so $h(w) = h(v \cdot a) = h(v) \cdot h(a)$ which reduces the value of $h(w)$ to known values. ∎

Exercise 4.3. *How many homomorphism exist with h such that $h(012) = 44444$, $h(102) = 444444$, $h(00) = 44444$ and $h(3) = 4$? Here two homomorphism are the same iff they have the same values for $h(0), h(1), h(2), h(3)$. Prove the answer: List the homomorphism to be counted and explain why there are not more.*

Exercise 4.4. *How many homomorphisms h exist with $h(012) = 44444$, $h(102) = 44444$, $h(0011) = 444444$ and $h(3) = 44$? Prove the answer: List the homomorphism to be counted and explain why there are not more.*

Theorem 4.5. *The homomorphic image of regular and context-free languages are regular and context-free, respectively.*

Proof. Let a regular / context-free grammar (N, Σ, P, S) for a language L be given and let Γ be the alphabet of all symbols which appear in some word of the form $h(a)$ with $a \in \Sigma$. One extends the homomorphism h to all members of N by defining $h(A) = A$ for all of them and one defines $h(P)$ as the set of all rules $h(l) \to h(r)$ where $l \to r$ is a rule in P; note that $h(l) = l$ in this case. Now $(N, \Gamma, h(P), S)$ is a new context-free grammar which generates $h(L)$; furthermore, if (N, Σ, P, S) is regular so is $(N, \Gamma, h(P), S)$.

First it is easy to verify that if all rules of P have only one non-terminal on the left side, so do those of $h(P)$; if all rules of P are regular, that is, either of the form $A \to w$ or of the form $A \to wB$ for non-terminals A, B and $w \in \Sigma^*$ then the image of the rule under h is of the form $A \to h(w)$ or $A \to h(w)B$ for a word $h(w) \in \Gamma^*$. Thus the transformation preserves the grammar to be regular or context-free, respectively.

Second one shows that if $S \Rightarrow^* v$ in the original grammar then $S \Rightarrow^* h(v)$ in the new grammar. The idea is to say that there are a number n and words $v_0 = S, v_1, \ldots v_n$ in $(N \cup \Sigma)^*$ such that $v_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_n = v$. Now one defines $w_m = h(v_m)$ for all m and proves that $S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = h(v)$ in the new grammar. So let $m \in \{0, 1, \ldots, n-1\}$ and assume that it is verified that $S \Rightarrow^* w_m$ in the new grammar. As $v_m \Rightarrow v_{m+1}$ in the old grammar, there are $x, y \in (\Sigma \cup N)^*$ and a rule $l \to h$ with $v_m = xly$ and $v_{m+1} = xry$. It follows that $w_m = h(x) \cdot h(l) \cdot h(y)$ and $w_{m+1} = h(x) \cdot h(r) \cdot h(y)$. Thus the rule $h(l) \to h(r)$ of the new grammar is applicable and $w_m \Rightarrow w_{m+1}$, that is, $S \Rightarrow^* w_{m+1}$ in the new grammar. Thus $w_n = h(v)$ is in the language generated by the new grammar.

Third one considers $w \in h(L)$. There are n and $w_0, w_1, \ldots, w_n$ such that $S = w_0$, $w = w_n$ and $w_m \Rightarrow w_{m+1}$ in the new grammar for all $m \in \{0, 1, \ldots, n-1\}$. Now one defines inductively $v_0, v_1, \ldots, v_n$ as follows: $v_0 = S$ and so $w_0 = h(v_0)$. Given now v_m with $h(v_m) = w_m$, the word w_m, w_{m+1} can be split into $\tilde{x}h(l)\tilde{y}$ and $\tilde{x}h(r)\tilde{y}$, respectively, for some rule $h(l) \to h(r)$ in $h(P)$. As h maps non-terminals to themselves, one can split v_m into $x \cdot l \cdot y$ such that $h(x) = \tilde{x}$ and $h(y) = \tilde{y}$. Now one defines v_{m+1} as $x \cdot r \cdot y$ and has that $w_{m+1} = h(v_{m+1})$ and $v_m \Rightarrow v_{m+1}$ by applying the rule $l \to r$ from P in the old grammar. It follows that at the end the so constructed sequence satisfies $v_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_n$ and $h(v_n) = w_n$. As w_n contains only terminals, v_n cannot contain any nonterminals and $v_n \in L$, thus $w_n \in h(L)$.

Thus the items Second and Third give together that $h(L)$ is generated by the grammar $(N, \Gamma, h(P), S)$ and the item First gave that this grammar is regular or context-free, respectively, as the given original grammar. ∎

Example 4.6. One can apply the homomorphisms also directly to regular expressions using the rules $h(L \cup H) = h(L) \cup h(H)$, $h(L \cdot H) = h(L) \cdot h(H)$ and $h(L^*) = (h(L))^*$. Thus one can move a homomorphism into the inner parts (which are the finite sets used in the regular expression) and then apply the homomorphism there.

So for the language $(\{0, 1\}^* \cup \{0, 2\}^*) \cdot \{33\}^*$ and the homomorphism which maps each symbol a to aa, one obtains the language $(\{00, 11\}^* \cup \{00, 22\}^*) \cdot \{3333\}^*$.

Exercise 4.7. *Consider the following statements for regular languages L:*

(a) $h(\emptyset) = \emptyset$;
(b) *If L is finite so is $h(L)$;*
(c) *If L has polynomial growth so has $h(L)$;*
(d) *If L has exponential growth so has $h(L)$.*

Which of these statements are true and which are false? Prove the answers. The rules from Example 4.6 can be used as well as the following facts: H^* has polynomial growth iff $H^* \subseteq \{u\}^*$ for some word u; if H, K have polynomial growth so do $H \cup K$ and $H \cdot K$.

Exercise 4.8. *Construct a context-sensitive language L and a homomorphism h such that L has polynomial growth and $h(L)$ has exponential growth.*

If one constructs regular expressions from automata or grammars, one uses a lot of Kleene stars, even nested into each other. However, if one permits the usage of homomorphism and intersections in the expression, one can reduce the usage of stars to the overall number of two. That intersections can save stars, can be seen by this example:

$$\begin{aligned} 00^* \cup 11^* \cup 22^* \cup 33^* &= (\{0,1,2,3\} \cdot \{00,11,22,33\}^* \cdot \{\varepsilon,0,1,2,3\}) \\ &\cap (\{00,11,22,33\}^* \cdot \{\varepsilon,0,1,2,3\}). \end{aligned}$$

The next result shows that adding in homomorphisms, the result can be used to represent arbitrary complex regular expressions using only two Kleene star sets and one homomorphism.

Theorem 4.9. *Let L be a regular language. Then there are two regular expressions σ, τ each containing only one Kleene star and some finite sets and concatenations and there is one homomorphism h such that L is the language given by the expression $h(\sigma \cap \tau)$.*

Proof. Assume that a nfa $(Q, \Gamma, \delta, s, F)$ recognises the language $L \subseteq \Gamma^*$. Now one makes a new alphabet Σ containing all triples (q, a, r) such that $a \in \Gamma$ and $q, r \in Q$ for which the nfa can go from q to r on symbol a. Let Δ contain all pairs $(q, a, r)(r, b, o)$ from $\Sigma \times \Sigma$ where the outgoing state of the first transition-triple is the incoming state of the second transition-triple. The regular expressions are now

$$\begin{aligned} \sigma = {} & \{(q,a,r) \in \Sigma\colon\ q = s \text{ and } r \in F\} \cup (\{(q,a,r) \in \Sigma : q = s\} \cdot \Delta^* \cdot \{(q,a,r)(r,b,o), \\ & (r,b,o) \in \Delta \cup \Sigma\colon\ o \in F\}); \end{aligned}$$

$$\begin{aligned} \tau = {} & \{(q,a,r) \in \Sigma\colon\ q = s \text{ and } r \in F\} \cup \{(q,a,r)(r,b,o) \in \Delta\colon\ q = s \text{ and } o \in F\} \cup \\ & (\{(q,a,r)(r,b,o) \in \Delta\colon\ q = s\} \cdot \Delta^* \cdot \{(q,a,r)(r,b,o), (r,b,o) \in \Delta \cup \Sigma\colon\ o \in F\}). \end{aligned}$$

Furthermore, the homomorphism h from Σ to Γ maps (q, a, r) to a for all $(q, a, r) \in \Sigma$. When allowing h and $\cap$ for regular expressions, one can describe L as follows: If L does not contain ε then L is $h(\sigma \cap \tau)$ else L is $h((\sigma \cup \{\varepsilon\}) \cap (\tau \cup \{\varepsilon\}))$.

The reason is that σ and τ both recognise runs of the nfa on words where the middle parts of the symbols in Σ represent the symbols read in Γ by the nfa and the other two parts are the states. However, the expression σ checks the consistency of the states (outgoing state of the last operation is ingoing state of the next one) only after reading an even number of symbols while τ checks the consistency after reading an odd number of symbols. In the intersection of the languages of σ and τ are then only those runs which are everywhere correct on the word. The homomorphism h translates the runs back into the words. ∎

Example 4.10: Illustrating Theorem 4.9. Let L be the language of all words which contain some but not all decimal digits. An nfa which recognises L has the states $\{s, q_0, q_1, \ldots, q_9\}$ and transitions (s, a, q_b) and (q_b, a, q_b) for all distinct $a, b \in \{0, 1, \ldots, 9\}$. Going to state q_b means that the digit b never occurs and if it would occur, the run would get stuck. All states are accepting.

The words 0123 and 228822 are in L and 0123456789 is not in L. For the word 0123, the run $(s, 0, q_4)\,(q_4, 1, q_4)\,(q_4, 2, q_4)\,(q_4, 3, q_4)$ is accepting and in both the languages generated by σ and by τ. The invalid run $(s, 0, q_4)\,(q_4, 1, q_4)\,(q_0, 2, q_0)\,(q_0, 3, q_0)$ would be generated by τ but not by σ, as σ checks that the transitions $(q_4, 1, q_4)\,(q_0, 2, q_0)$ match.

As the language contains the empty word, it would be generated by $h((\sigma \cup \{\varepsilon\}) \cap (\tau \cup \{\varepsilon\}))$.

Homomorphisms allow to map certain symbols to ε; this permits to make the output of a grammar shorter. As context-sensitive languages are produced by grammars which generate words getting longer or staying the same in each step of the derivation, there is a bit a doubt what happens when output symbols of a certain type get erased in the process of making them. Indeed, one can use this method in order to show that any language L generated by some grammar is the homomorphic image of a context-sensitive language; thus the context-sensitive languages are not closed under homomorphisms.

Theorem 4.11. *Every recursively enumerable language, that is, every language generated by some grammar, is a homomorphic image of a context-sensitive language.*

Proof. Assume that the alphabet is $\{1, 2, \ldots, k\}$ and that 0 is a digit not occurring in any word of L. Furthermore, assume that $(N, \{1, 2, \ldots, k\}, P, S)$ is a grammar generating the language L; without loss of generality, all rules $l \to r$ satisfy that $l \in N^+$; this can easily be achieved by introducing a new non-terminal A for each terminal a, replacing a in all rules by A and then adding the rule $A \to a$.

Now one constructs a new grammar $(N, \{0, 1, 2, \ldots, k\}, P', S)$ as follows: For each rule $l \to r$ in P, if $|l| \leq |r|$ then P' contains the rule $l \to r$ unchanged else P' contains the rule $l \to r^{|l|}$. Furthermore, P' contains for every $A \in N$ the rule $0A \to A0$ which permits to move every 0 towards the end along non-terminals. There are no other rules in P' and the grammar is context-sensitive. Let H be the language generated by this new grammar.

Now define $h(0) = \varepsilon$ and $h(a) = a$ for every other $a \in N \cup \{1, 2, \ldots, k\}$. It will be shown that $L = h(H)$.

First one considers the case that $v \in L$ and looks for a $w \in H$ with $h(w) = v$. There is a derivation $v_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_n$ of v with $v_0 = S$ and $v_n = v$. Without loss of generality, all rules of the form $A \to a$ for a non-terminal A and terminal a are applied after all other rules are done. Now it will be shown by induction that there are numbers $\ell_0 = 0$, $\ell_1, \ldots, \ell_n$ such that all $w_m = v_m 0^{\ell_m}$ satisfy $w_0 \Rightarrow^* w_1 \Rightarrow^* \ldots \Rightarrow^* w_n$. Note that $w_0 = S$, as $\ell_0 = 0$. Assume that w_m is defined. There is a rule $l \to r$. If r is a terminal then l is one non-terminal and furthermore the rule $l \to r$ also exists in P', thus one applies the same rule to the same position in w_m and let $\ell_{m+1} = \ell_m$ and has that $w_m \Rightarrow w_{m+1}$. If r is not a non-terminal then for the rule $l \to r$ in P there might be some rule $l \to r0^\kappa$ in P' and $v_m = xly \Rightarrow v_{m+1} xry$ in the old grammar and $w_m = xly0^{\ell_m} \Rightarrow xr0^\kappa y0^{\ell_m} \Rightarrow^* xry0^{\kappa+\ell_m} = xry0^{\ell_{m+1}} = w_{m+1}$, where one has to make the definition $\ell_{m+1} = \kappa + \ell_m$ and where the step $xr0^\kappa y0^{\ell_m} \Rightarrow^* xry0^{\kappa+\ell_m}$ is possible as no other terminals than 0 are generated so far. this rule is applied before generating any other non-terminal than 0 and therefore one has $v_m 0^{\ell_m} \Rightarrow v_{m+1}$. Thus $w_m \Rightarrow^* w_{m+1}$ in the grammar for H. It follows that $w_n \in H$ and $h(w_n) = v$.

Now assume that $v = h(w)$ and $w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$ is a derivation of w in the grammar for H. Let $v_m = h(w_m)$ for all m. Note that $h(v_0) = S$. For each $m < n$, if w_{m+1} is obtained from w_m by exchanging the position of a non-terminal and 0 then $h(w_{m+1}) = h(w_m)$ and $v_m \Rightarrow^* v_{m+1}$. Otherwise $w_m = xly$ and $w_{m+1} = xr0^\kappa y$ for some x, y and rule $l \to r0^\kappa$ in P' (where $0^\kappa = \varepsilon$ is possible). Now $v_m = h(w_m) = h(x) \cdot l \cdot h(y)$ and $v_{m+1} = h(w_{m+1}) = h(x) \cdot r \cdot h(y)$, thus $v_m \Rightarrow v_{m+1}$ in the grammar for L. It follows that $v_0 \Rightarrow^* v_1 \Rightarrow^* \ldots \Rightarrow^* v_n$ and $v_n = h(w) \in L$.

The last two parts give that $L = h(H)$ and the construction of the grammar ensured that H is a context-sensitive language. Thus L is the homomorphic image of a context-sensitive language. ∎

Proposition 4.12. *If a grammar (N, Σ, P, S) generates a language L and h is a homomorphism from Σ^* to Σ^* and $S', T', U' \notin N$ then the grammar given as $(N \cup \{S', T', U'\}, \Sigma, P', S')$ with $P' = P \cup \{S' \to T'SU', T'U' \to \varepsilon\} \cup \{T'a \to h(a)T' : a \in \Sigma\}$ generates $h(L)$.*

Proof Idea. If $a_1, a_2, \ldots, a_n \in \Sigma$ then $T'a_1a_2 \ldots a_n \Rightarrow^* h(a_1)h(a_2) \ldots h(a_n)$. Thus if $S \Rightarrow w$ in the original grammar then $S' \Rightarrow T'SU' \Rightarrow^* T'wU' \Rightarrow^* h(w)T'U' \Rightarrow h(w)$ in the new grammar and one can also show that this is the only way which permits to derive terminal words in the new grammar. ∎

Exercise 4.13. *Let $h(0) = 1$, $h(1) = 22$, $h(2) = 333$. What are $h(L)$ for the following languages L:*

(a) $\{0, 1, 2\}^*$;

(b) $\{00, 11, 22\}^* \cap \{000, 111, 222\}^*$;
(c) $(\{00, 11\}^* \cup \{00, 22\}^* \cup \{11, 22\}^*) \cdot \{011222\}$;
(d) $\{w \in \{0, 1\}^* : w$ *has more* 1*s than it has* 0*s*$\}$.

Exercise 4.14. *Let* $h(0) = 3$, $h(1) = 4$, $h(2) = 334433$. *What are* $h(L)$ *for the following languages* L:

(a) $\{0, 1, 2\}^*$;
(b) $\{00, 11, 22\}^* \cap \{000, 111, 222\}^*$;
(c) $(\{00, 11\}^* \cup \{00, 22\}^* \cup \{11, 22\}^*) \cdot \{011222\}$;
(d) $\{w \in \{0, 1\}^* : w$ *has more* 1*s than it has* 0*s*$\}$.

The next series of exercises deal with homomorphisms between number systems. In general, it is for example known that the homomorphism h given by $h(0) = 0000, h(1) = 0001, h(2) = 0010, \ldots, h(F) = 1111$ translate numbers from the hexadecimal system into binary numbers preserving their value. However, one conventions are not preserved: there might be leading zeroes introduced and the image of $1F$ is 00011111 rather than the correct 11111. The following translations do not preserve the value, as this is only possible when translating numbers from a base system p^n to the base system p for some number p. However, they try to preserve some properties. The exercises investigate to which extent various properties can be preserved simultaneously.

Exercise 4.15. *Let a homomorphism* $h : \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \to \{0, 1, 2, 3\}^*$ *be given by the equations* $h(0) = 0$, $h(1) = h(4) = h(7) = 1$, $h(2) = h(5) = h(8) = 2$, $h(3) = h(6) = h(9) = 3$. *Interpret the images of* h *as quaternary numbers (numbers of base four, so* 12321 *represents* 1 *times two hundred fifty six plus* 2 *times sixty four plus* 3 *times sixteen plus* 2 *times four plus* 1*). Prove the following:*

- *Every quaternary number is the image of a decimal number without leading zeroes;*
- *A decimal number* w *has leading zeroes iff the quaternary number* $h(w)$ *has leading zeroes;*
- *A decimal number* w *is a multiple of three iff the quaternary number is a multiple of three.*

Exercise 4.16. *Consider any homomorphism* $h : \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \to \{0, 1\}^*$ *such that*

- *$h(w)$ has leading zeroes iff w has;*
- *$h(0) = 0$;*
- *all binary numbers (without leading zeroes) are in the range of h.*

Answer the following questions:

(a) *Can h be chosen such that the above conditions are true and, furthermore, the decimal number w is a multiple of two iff the binary number $h(w)$ is a multiple of two?*
(b) *Can h be chosen such that the above conditions are true and, furthermore, the decimal number w is a multiple of three iff the binary number $h(w)$ is a multiple of three?*
(c) *Can h be chosen such that the above conditions are true and, furthermore, the decimal number w is a multiple of five iff the binary number $h(w)$ is a multiple of five?*

If h can be chosen as desired then list this h else prove that such a homomorphism h cannot exist.

Exercise 4.17. *Construct a homomorphism $h : \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \to \{0, 1\}^*$ such that for every w the number $h(w)$ has never leading zeroes and the remainder of the decimal number w when divided by nine is the same as the remainder of the binary number $h(w)$ when divided by nine.*

Another way to represent is the Fibonacci number system. Here one let $a_0 = 1$, $a_1 = 1$, $a_2 = 2$ and, for all n, $a_{n+2} = a_n + a_{n+1}$. Now one can write every number as the sum of non-neighbouring Fibonacci numbers: That is for each non-zero number n there is a unique string $b_m b_{m-1} \ldots b_0 \in (10^+)^+$ such that

$$n = \sum_{k=0,1,\ldots,m} b_k \cdot a_k$$

and the next exercise is about this numbering.

Exercise 4.18. *Construct a homomorphism $h : \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \to \{0, 1\}^*$ such that $h(0) = 0$ and the image of all decimal numbers (without leading zeroes) is the regular set $\{0\} \cup (10^+)^+$. Furthermore, show that all h satisfying the above condition also satisfy the following statement: For every $p > 1$ there is a decimal number w (without leading zeroes) such that (w is a multiple of p iff $h(w)$ is not a multiple of p). In other word, the property of being a multiple of p is not preserved by h for any $p > 1$.*

Description 4.19: Inverse Homomorphism. Assume that $h : \Sigma^* \Rightarrow \Gamma^*$ is a homomorphism and $L \subseteq \Gamma^*$ is some language. Then $K = \{u \in \Sigma^* : h(u) \in H\}$ is called the inverse image of L with respect to the homomorphism h. This inverse image K is also denoted as $h^{-1}(L)$. The following rules are valid for h^{-1}:

(a) $h^{-1}(L) \cap h^{-1}(H) = h^{-1}(L \cap H)$;
(b) $h^{-1}(L) \cup h^{-1}(H) = h^{-1}(L \cup H)$;
(c) $h^{-1}(L) \cdot h^{-1}(H) \subseteq h^{-1}(L \cdot H)$;
(d) $h^{-1}(L)^* \subseteq h^{-1}(L^*)$.

One can see (a) as follows: If $h(u) \in L$ and $h(u) \in H$ then $h(u) \in L \cap H$, thus $u \in h^{-1}(L) \cap h^{-1}(H)$ implies $u \in h^{-1}(L \cap H)$. If $u \in h^{-1}(L \cap H)$ then $h(u) \in L \cap H$ and $u \in h^{-1}(L) \cap h^{-1}(H)$.

For (b), if $h(u) \in L$ or $h(u) \in H$ then $h(u) \in L \cup H$, thus $u \in h^{-1}(L) \cup h^{-1}(H)$ implies $u \in h^{-1}(L \cup H)$. If $u \in h^{-1}(L \cup H)$ then $h(u) \in L \cup H$ and $u \in h^{-1}(L)$ or $u \in h^{-1}(H)$, so $u \in h^{-1}(L) \cup h^{-1}(H)$.

For (c), note that if $u \in h^{-1}(L) \cdot h^{-1}(H)$ then there are v, w with $u = vw$ and $v \in h^{-1}(L)$ and $w \in h^{-1}(H)$. Thus $h(u) = h(v) \cdot h(w) \in L \cdot H$ and $u \in h^{-1}(L \cdot H)$. However, if $\Sigma = \{0\}$ and $h(0) = 00$ then $h^{-1}(\{0\}) = \emptyset$ while $h^{-1}(\{0\} \cdot \{0\}) = \{0\}$ which differs from $\emptyset \cdot \emptyset$. Therefore the inclusion can be proper.

For (d), if $v_1, v_2, \ldots, v_n \in h^{-1}(L^*)$ then $v_1 v_2 \ldots v_n \in h^{-1}(L^*)$ as well; thus $h^{-1}(L^*)$ is a set of the form H^* which contains $h^{-1}(L)$. However, the inclusion can be proper: Using the h of (c), $h^{-1}(\{0\}) = \emptyset$, $(h^{-1}(\{0\}))^* = \{\varepsilon\}$ and $h^{-1}(\{0\}^*) = \{0\}^*$.

Theorem 4.20. *If L is on the level k of the Chomsky hierarchy and h is a homomorphism then $h^{-1}(L)$ is also on the level k of the Chomsky hierarchy.*

Proof for the regular case. Assume that $L \subseteq \Gamma^*$ is recognised by a dfa $(Q, \Gamma, \gamma, s, F)$ and that $h : \Sigma^* \to \Gamma^*$ is a homomorphism. Now one constructs a new dfa $(Q, \Sigma, \delta, s, F)$ with $\delta(q, a) = \gamma(q, h(a))$ for all $q \in Q$ and $a \in \Sigma$. One can show by an induction that when the input word is w then the new dfa is in the state $\delta(s, w)$ and that this state is equal to the state $\gamma(s, h(w))$ and therefore the new automaton accepts w iff the old automaton accepts $h(w)$. It follows that w is accepted by the new dfa iff $h(w)$ is accepted by the old automaton iff $h(w) \in L$ iff $w \in h^{-1}(L)$. Thus h^{-1} is regular, as witnessed by the new automaton. ∎

Exercise 4.21. *Let $h : \{0, 1, 2, 3\}^* \to \{0, 1, 2, 3\}^*$ be given by $h(0) = 00$, $h(1) = 012$, $h(2) = 123$ and $h(3) = 1$ and let L contain all words containing exactly five 0s and at least one 2. Construct a complete dfa recognising $h^{-1}(L)$.*

5 Normalforms and Algorithms

For context-free languages, there are various normal forms which can be used in order to make algorithms or carry out certain proofs. These two normal forms are the following ones.

Definition 5.1: Normalforms. *Consider a context-free grammar* (N, Σ, P, S) *with the following basic properties: if* $S \Rightarrow^* \varepsilon$ *then* $S \to \varepsilon$ *occurs in* P *and no rule has any occurrence of* S *on the right side; there is no rule* $A \to \varepsilon$ *for any* $A \neq S$*;*

The grammar is in **Chomsky Normal Form** *in the case that every rule (except perhaps* $S \to \varepsilon$*) is either of the form* $A \to a$ *or of the form* $A \to BC$ *for some* $A, B, C \in N$ *and terminal* $a \in \Sigma$*.*

The grammar is in **Greibach Normal Form** *in the case that every rule (except perhaps* $S \to \varepsilon$*) has a right hand side from* ΣN^**.*

Algorithm 5.2: Chomsky Normal Form. There is an algorithm which transforms any given context-free grammar (N_0, Σ, P_0, S) into a new grammar (N_4, Σ, P_4, S') in Chomsky Normal Form. Assume that the grammar produces at least one word (otherwise the algorithm will end up with a grammar with an empty set of non-terminals).

1. **Dealing with** ε**:** Let $N_1 = N_0 \cup \{S'\}$ for a new non-terminal S'; Initialise $P_1 = P_0 \cup \{S' \to S\}$;
   While there are $A, B \in N_1$ and $v, w \in (N_1 \cup \Sigma)^*$ with $A \to vBw, B \to \varepsilon$ in P_1 and $A \to vw$ not in P_1 Do Begin $P_1 = P_1 \cup \{A \to vw\}$ End;
   Remove all rules $A \to \varepsilon$ for all $A \in N_0$ from P_1, that is, for all $A \neq S'$;
   Keep N_1, P_1 fixed from now on and continue with grammar (N_1, Σ, P_1, S');

2. **Dealing with single terminal letters:** Let $N_2 = N_1$ and $P_2 = P_1$;
   While there are a letter $a \in \Sigma$ and a rule $A \to w$ in P_2 with $w \neq a$ and a occurring in w
   Do Begin Choose a new non-terminal $B \notin N_2$;
   Replace in all rules in P_2 all occurrences of a by B;
   update $N_2 = N_2 \cup \{B\}$ and add rule $B \to a$ to P_2 End;
   Continue with grammar (N_2, Σ, P_2, S');

3. **Breaking long ride hand sides:** Let $N_3 = N_2$ and $P_3 = P_2$;
   While there is a rule of the form $A \to Bw$ in P_3 with $A, B \in N_3$ and $w \in N_3 \cdot N_3^+$
   Do Begin Choose a new non-terminal $C \notin N_3$ and let $N_3 = N_3 \cup \{C\}$;
   Add the rules $A \to BC, C \to w$ into P_3 and remove the rule $A \to Bw$ End;
   Continue with grammar (N_3, Σ, P_3, S');

4. Removing rules $\mathbf{A \to B}$: Make a table of all $(A, B), (A, a)$ such that $A, B \in N_3$, $a \in \Sigma$ and, in the grammar (N_3, Σ, P_3, S'), $A \Rightarrow^* B$ and $A \Rightarrow^* a$, respectively;
Let $N_4 = N_3$ and P_4 contain the following rules:
$S' \to \varepsilon$ in the case that this rule is in P_3;
$A \to a$ in the case that (A, a) is in the table;
$A \to BC$ in the case that there is $D \to EF$ in P_3 with $(A, D), (E, B), (F, C)$ in the table;
The grammar (N_4, Σ, P_4, S') is in Chomsky Normalform.

Example 5.3. Consider the grammar $(\{S\}, \{0, 1\}, \{S \to 0S0|1S1|00|11\}, S)$ making all palindromes of even length; this language generates all non-terminals of even length. A grammar in Chomsky Normal Form for this language needs much more non-terminals. The set of non-terminals is $\{S, T, U, V, W\}$, the alphabet is $\{0, 1\}$, the start symbols is S and the rules are $S \to TV|UW$, $T \to VS|0$, $U \to WS|1$, $V \to 0$, $W \to 1$. The derivation $S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 010010$ in the old grammar is equivalent to $S \Rightarrow TV \Rightarrow VSV \Rightarrow 0SV \Rightarrow 0S0 \Rightarrow 0UW0 \Rightarrow 0WSW0 \Rightarrow 0WS10 \Rightarrow 01S10 \Rightarrow 01TV10 \Rightarrow 010V10 \Rightarrow 010010$ in the new grammar.

Exercise 5.4. *Bring the grammar* $(\{S, T\}, \{0, 1\}, \{S \to TTTT, T \to 0T1|\varepsilon\}, S)$ *into Chomsky Normal Form.*

Exercise 5.5. *Bring the grammar* $(\{S, T\}, \{0, 1\}, \{S \to ST|T, T \to 0T1|01\}, S)$ *into Chomsky Normal Form.*

Exercise 5.6. *Bring the grammar* $(\{S\}, \{0, 1\}, \{S \to 0SS11SS0, 0110\}, S)$ *into Chomsky Normal Form.*

Algorithm 5.7: Removal of Useless Non-Terminals. When given a grammar (N_0, Σ, P_0, S) in Chomsky Normal Form, one can construct a new grammar (N_2, Σ, P_2, S) which does not have useless non-terminals, that is, every non-terminal can be derived into a word of terminals and every non-terminal can occur in some derivation.

1. Removing non-terminating non-terminals: Let N_1 contain all $A \in N_0$ for which there is a rule $A \to a$ or a rule $A \to \varepsilon$ in P_0;
While there is a rule $A \to BC$ in P_0 with $A \in N_0 - N_1$ and $B, C \in N_1$ Do Begin $N_1 = N_1 \cup \{A\}$ End;
Let P_1 be all rules $A \to w$ in P_0 such that $A \in N_1$ and $w \in N_1 \cdot N_1 \cup \Sigma \cup \{\varepsilon\}$;
If $S \notin N_1$ then terminate with empty grammar else continue with grammar (N_1, Σ, P_1, S).

2. Selecting all reachable non-terminals: Let $N_2 = \{S\}$;
While there is a rule $A \to BC$ in P_1 with $A \in N_2$ and $\{B, C\} \not\subseteq N_2$
Do Begin $N_2 = N_2 \cup \{B, C\}$ End;
Let P_2 contain all rules $A \to w$ in P_1 with $A \in N_2$ and $w \in N_2 \cdot N_2 \cup \Sigma \cup \{\varepsilon\}$;
The grammar (N_2, Σ, P_2, S) does not contain any useless non-terminal.

Quiz 5.8. *Consider the grammar with terminal symbol* 0 *and non-terminal symbols* $Q, R, S, T, U, V, W, X, Y, Z$ *and rules* $S \to TU|UV, T \to UT|TV|TW, R \to VW|QQ|0, Q \to 0, U \to VW|WX, V \to WX|XY|0, W \to XY|YZ|0$ *and start symbol* S*. Determine the set of reachable and terminating non-terminals.*

Exercise 5.9. *Consider the grammar*

$(\{S_0, S_1, \ldots, S_9\}, \{0\}, \{S_0 \to S_0S_0,\ S_1 \to S_2S_3,\ S_2 \to S_4S_6|0,\ S_3 \to S_6S_9,\ S_4 \to S_8S_2,\ S_5 \to S_0S_5,\ S_6 \to S_2S_8,\ S_7 \to S_4S_1|0,\ S_8 \to S_6S_4|0,\ S_9 \to S_8S_7\},\ S_1)$.

Determine the set of reachable and terminating non-terminals and explain the steps on the way to this set. What is the shortest word generated by this grammar?

Exercise 5.10. *Consider the grammar*

$(\{S_0, S_1, \ldots, S_9\},\ \{0\},\ \{S_0 \to S_1S_1,\ S_1 \to S_2S_2,\ S_2 \to S_3S_3,\ S_3 \to S_0S_0|S_4S_4,\ S_4 \to S_5S_5,\ S_5 \to S_6S_6|S_3S_3,\ S_6 \to S_7S_7|0,\ S_7 \to S_8S_8|S_7S_7,\ S_8 \to S_7S_6|S_8S_6,\ S_9 \to S_7S_8|0\},\ S_1)$.

Determine the set of reachable and terminating non-terminals and explain the steps on the way to this set. What is the shortest word generated by this grammar?

Algorithm 5.11: Emptyness Check. The above algorithms can also be used to check whether a context-free grammar produces any word. The algorithm works indeed for any context-free grammar by using Algorithm 5.2 to make the grammar into Chomsky Normal Form and then Algorithm 5.7 to remove the useless symbols. A direct check would be the following for a context-free grammar (N, Σ, P, S):

Initialisation: Let $N' = \emptyset$;

Loop: While there are $A \in N - N'$ and a rule $A \to w$ with $w \in (N' \cup \Sigma)^*$
Do Begin $N' = N' \cup \{A\}$ End;

Decision: If $S \notin N'$ then the language of the grammar is empty else the language of the grammar contains some word.

Algorithm 5.12: Finiteness Check. One can also check whether a language in Chomsky Normal Form generates an infinite set. This algorithm is an extended version of the previous Algorithm 5.11: In the first loop, one determines the set N' of non-terminals which can be converted into a word (exactly as before), in the second loop one determines for all members $A \in N'$ the set $N''(A)$ of non-terminals which can be obtained from N' in a derivation which needs more than one step and which only uses rules where all members on the right side are in N'. If such a non-terminal A satisfies $A \in N''(A)$ then one can derive infinitely many words from A; the same applies if there is $B \in N''(A)$ with $B \in N''(B)$. Thus the algorithm looks as follows:

Initialisation 1: Let $N' = \emptyset$;

Loop 1: While there are $A \in N - N'$ and a rule $A \to w$ with $w \in (N' \cup \Sigma)^*$
Do Begin $N' = N' \cup \{A\}$ End;

Initialisation 2: For all $A \in N$, let $N''(A) = \emptyset$;

Loop 2: While there are $A, B, C, D \in N'$ and a rule $B \to CD$ with $B \in N''(A) \cup \{A\}$ and ($C \notin N''(A)$ or $D \notin N''(A)$)
Do Begin $N''(A) = N''(A) \cup \{C, D\}$ End;

Decision: If there is $A \in N''(S) \cup \{S\}$ with $A \in N''(A)$ then the language of the grammar is infinite else it is finite.

Exercise 5.13. *The checks whether a grammar in Chomsky Normal Form generates the empty set or a finite set can be implemented to run in polynomial time. However, for non-empty grammars, these checks do not output an element witnessing that the language is non-empty. If one adds the requirement to list such an element completely (that is, all its symbols), what is the worst time complexity of this algorithm: polynomial in n, exponential in n, double exponential in n? Give reasons for the answer. Here n is the number of non-terminals in the grammar, note that the number of rules is then also limited by $O(n^3)$.*

Exercise 5.14. *Consider the grammar $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ with P consisting of the rules $S \to TT$, $T \to UU$, $U \to VW|WV$, $V \to 0$, $W \to 1$. How many words does the grammar generate: (a) None, (b) One, (c) Two, (d) Three, (e) Finitely many and at least four, (f) Infinitely many?*

Exercise 5.15. *Consider the grammar $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ with P consisting of the rules $S \to ST$, $T \to TU$, $U \to UV$, $V \to VW$, $W \to 0$. How many words does the grammar generate: (a) None, (b) One, (c) Two, (d) Three, (e) Finitely many and at least four, (f) Infinitely many?*

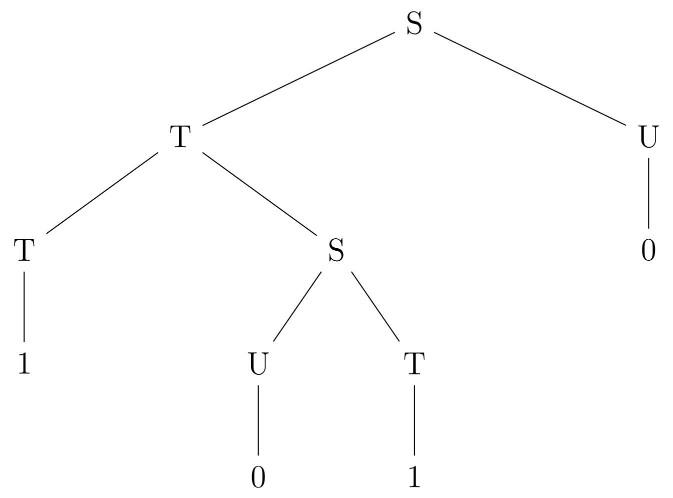
Exercise 5.16. *Consider the grammar* $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ *with* P *consisting of the rules* $S \to UT|TU|2$, $T \to VV$, $U \to WW$, $V \to 0$, $W \to 1$. *How many words does the grammar generate: (a) None, (b) One, (c) Two, (d) Three, (e) Finitely many and at least four, (f) Infinitely many?*

Exercise 5.17. *Consider the grammar* $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ *with* P *consisting of the rules* $S \to SS|TT|UU$, $T \to VV$, $U \to WW$, $V \to 0$, $W \to WW$. *How many words does the grammar generate: (a) None, (b) One, (c) Two, (d) Three, (e) Finitely many and at least four, (f) Infinitely many?*

Description 5.18: Derivation Tree. A derivation tree is a representation of a derivation of a word w by a context-free grammar such that each node in a tree is labeled with a non-terminal A and the successors of the node in the tree are those symbols, each in an extra node, which are obtained by applying the rule in the derivation to the symbol A. The leaves contain the letters of the word w and the root of the tree contains the start symbol S. For example, consider the grammar

$$(\{S, T, U\}, \{0, 1\}, \{S \to SS|TU|UT, U \to 0|US|SU, T \to 1|TS|ST\}, S).$$

Now the derivation tree for the derivation $S \Rightarrow TU \Rightarrow TSU \Rightarrow TUTU \Rightarrow 1UTU \Rightarrow 10TU \Rightarrow 101U \Rightarrow 1010$ can be the following:



However, this tree is not unique, as it is not clear whether in the derivation step $TU \Rightarrow TSU$ the rule $T \to TS$ was applied to T or $U \to SU$ was applied to U. Thus derivation trees can be used to make it clear how the derivation was obtained. On the other hand, derivation trees are silent about the order in which derivations are applied to nodes which are not above or below each other; this order is, however, also not relevant for the result obtained. For example, once the derived word has reached the length 4, it is irrelevant for the word obtained in which order one converts the non-terminals into terminals.

Exercise 5.19. *For the grammar from Description 5.18, how many derivation trees are there to derive* 011001*?*

Exercise 5.20. *For the grammar from Description 5.18, how many derivation trees are there to derive* 000111*?*

Exercise 5.21. *Consider the grammar*

$$(\{S,T\},\{0,1,2\},\{S \to TT, T \to 0T1|2\},S).$$

Draw the derivation tree for the derivation of 00211021 *in this grammar and prove that for this grammar, every word in the language generated has a unique derivation tree. Note that derivation trees are defined for all context-free grammars and not only for those in Chomsky Normal Form.*

The concept of the Chomsky Normal Form and of a Derivation Tree permit to prove the version of the traditional pumping lemma for context-free languages; this result was stated before, but the proof was delayed to this point.

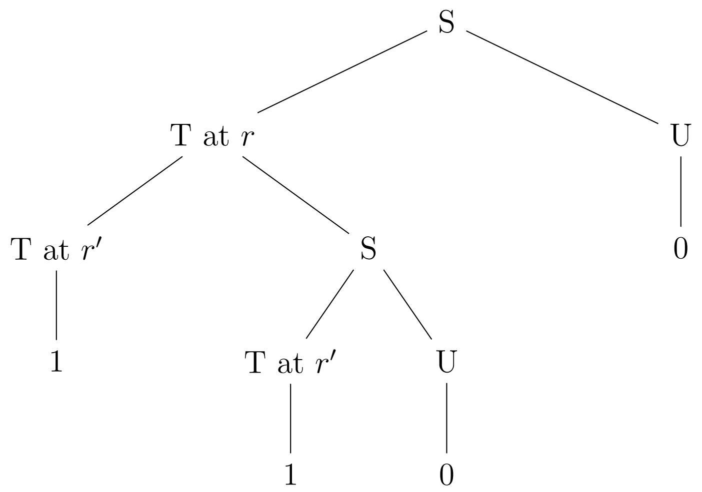
Theorem 1.35 (b). Let $L \subseteq \Sigma^*$ be an infinite context-free language generated by a grammar (N,Σ,P,S) in Chomsky Normal Form with h non-terminals. Then the constant $k = 2^{h+1}$ satisfies that for every $u \in L$ of length at least k there is a representation $vwxyz = u$ such that $|wxy| \leq k$, ($w \neq \varepsilon$ or $y \neq \varepsilon$) and $vw^\ell xy^\ell z \in L$ for all $\ell \in \mathbb{N}$.

Proof. Assume that $u \in L$ and $|u| \geq k$. Let R be a derivation tree for the derivation of u from S. If there is no branch of the tree R in which a non-terminal appears twice then each branch consists of at most h branching nodes and the number of leaves of the tree is at most $2^h < k$. Thus $|u| \geq k$ would be impossible. Note that the leaves in the tree have terminals in their nodes and the inner nodes have non-terminals in their node. For inner nodes r, let $A(r)$ denote the non-terminal in their node.

Thus there must be nodes $r \in R$ for which the symbol $A(r)$ equals to $A(r')$ for some descendant r'. By taking r with this property to be as distant from the root as possible, one has that there are no descendant r' of r and r'' of r' such that $A(r') = A(r'')$. Thus, each descendant r' of r has at most 2^h leaves descending from r' and r has at most $k = 2^{h+1}$ descendants. Now, if one terminalises the derivations except for what comes from $A(r)$ and the descendant $A(r')$ with $A(r') = A(r)$, one can split the word u into v, w, x, y, z such that $S \Rightarrow^* vA(r)z \Rightarrow^* vwA(r')yz \Rightarrow^* vwxyz$ and one has also that $A(r) \Rightarrow^* wA(r')y = wA(r)y$ and $A(r') \Rightarrow^* x$.

These observations permit to conclude that $S \Rightarrow^* vw^\ell xy^\ell z$ for all $\ell \in \mathbb{N}$. As $A(r) \Rightarrow wxy$ and the branches have below r at most h non-terminals on the branch, each such branch has at most $h+1$ branching nodes starting from r and the word part in u generated below r satisfies $|wxy| \leq 2^{h+1} = k$. Furthermore, only one of the two children of r can generate the part which is derived from $A(r')$, thus at least one of w, y must be non-empty. Thus the length constraints of the pumping lemma are satisfied as required. ∎

Example of a Derivation Tree for Proof. The following derivation tree shows an example on how r and r' are chosen. The choice is not always unique.



The grammar is the one from Description 5.18 and the tree is for deriving the word 1100. Both symbols S and T are repeated after their first appearance in the tree; however, T occurs later than S and so the node r is the node where T appears for the first time in the derivation tree. Now both descendants of r which have the symbol T in the node can be chosen as r' (as indicated). If one chooses the first, one obtains that all words of the form $1(10)^\ell 0$ are in the language generated by the grammar; as one would have that $S \Rightarrow^* T0 \Rightarrow^* T(10)^\ell 0 \Rightarrow^* 1(10)^\ell 0$. If one choose the second, one obtains that all words of the form $1^\ell 10^\ell 0$ are in the language generated by the grammar, as one would have that $S \Rightarrow^* T0 \Rightarrow^* 1^\ell T0^\ell 0 \Rightarrow 1^\ell 10^\ell 0$. In both cases, ℓ can be any natural number.

Ogden's Lemma is not imposing a length-constraint but instead permitting to mark symbols. Then the word will be split such that some but not too many of the marked symbols go into the pumped part. This allows to influence where the pump is.

Theorem 5.22: Ogden's Pumping Lemma [58]. *Let $L \subseteq \Sigma^*$ be an infinite context-free language generated by a grammar (N, Σ, P, S) in Chomsky Normal Form with h non-terminals. Then the constant $k = 2^{h+1}$ satisfies that for every $u \in L$ with at least k marked symbols, there is a representation $vwxyz = u$ such that wxy contains at most k marked symbols, wy contains at least 1 marked symbol and $vw^\ell xy^\ell z \in L$ for all $\ell \in \mathbb{N}$.*

Proof. Assume that $u \in L$ and $|u| \geq k$. Let R be a derivation tree for the derivation of u from S and assume that at least k symbols in u are marked. Call a node r in the tree a marked branching node iff marked symbols can be reached below both immediate successors of the node. If there is no branch of the tree R in which a non-terminal appears twice on marked branching positions then each branch contains at most h marked branching nodes and the number of leaves with marked symbols of the tree is at most $2^h < k$. Thus $u \geq k$ could not have k marked symbols.

Thus there must be marked branching nodes $r \in R$ for which the symbol $A(r)$ equals to $A(r')$ for some descendant r' which is also a marked branching node. By taking r with this property to be as distant from the root as possible, one has that there are no marked branching descendant r' of r and no marked branching descendant r'' of r' such that $A(r') = A(r'')$. Thus, each marked branching descendant r' of r has at most 2^h marked leaves descending from r' and r has at most $k = 2^{h+1}$ descendants which are marked leaves. Now, if one terminalises the derivations except for what comes from $A(r)$ and the descendant $A(r')$ with $A(r') = A(r)$, one can split the word u into v, w, x, y, z such that $S \Rightarrow^* vA(r)z \Rightarrow^* vwA(r')yz \Rightarrow^* vwxyz$ and one has also that $A(r) \Rightarrow^* wA(r')y = wA(r)y$ and $A(r') \Rightarrow^* x$.

These observations permit to conclude that $S \Rightarrow^* vw^\ell xy^\ell z$ for all $\ell \in \mathbb{N}$. As $A(r) \Rightarrow wxy$ and the branches have below r at most h non-terminals on marked branching nodes, each such branch has at most $h+1$ marked branching nodes starting from r and the word part wxy in u generated below r satisfies that it contains at most $2^{h+1} = k$ many marked leaves. Furthermore, only one of the two children of r can generate the part which is derived from $A(r')$, thus at least one of w, y must contain some marked symbols. Thus the length constraints of Ogden's Pumping Lemma are satisfied as required. ∎

Example 5.23. *Let L be the language of all words $w \in 1^+(0^+1^+)^+$ where no two runs of zeroes have the same length. So $100010011 \in L$ and $11011011001 \notin L$. Now L satisfies the traditional pumping lemma for context-free languages but not Ogden's Pumping Lemma.*

Proof. First one shows that L satisfies Corollary 1.36 with pumps of the length of one symbol; this then implies that also Theorem 1.35 (b) is satisfied, as there only the length of the pumps with the part in between is important but not their position in the full word. Let $u \in L$ and $|u| > 3$. If 11 occurs in u then one can pump the first 1 which neighbours to another 1, as pumping this 1 does not influence the number and lengths of the runs of zeroes in the word u. If 11 does not occur in u and there is only one run of zeroes, so $u \in 10^+1$, then one pumps the first of these zeroes; note that $11 \in L$ and thus if $u = 10^{h+1}1$ then $10^*0^h1 \subseteq L$. If there are several runs and 11 does not occur in u then one pumps the border between the longest run of zeroes and some neighbouring run of zeroes; if this border is omitted then the run of zeroes becomes longer and is different from all others in length; if this border is repeated, the number and lengths of the runs of zeroes is not modified. For example, if $u = 1001010001$ then $100101^*0001 \subseteq L$, as in the case that the pump is omitted the resulting word 100100001 is in L as well, as the longest run of zeroes is fusionated with another run of zeroes.

In order to see that L does not satisfy Ogden's Lemma, one considers the word

$$u = 1010^2 10^3 1 \ldots 10^{4k-1} 10^{4k} 1$$

and one marks the k zeroes in the subword 10^k1 of u. Now consider any splitting $vwxyz$ of u.

If w is in $\{0,1\}^* - \{0\}^* - \{1\}^*$ then ww contains a subword of the form 10^h1 and xw^4xy^4z contains this subword at least twice. Thus w cannot be chosen from this set; similarly for y.

If w, y are both in $\{1\}^*$ then none of the marked letters is pumped and therefore this is also impossible.

If $w \in \{0\}^+$ and $y \in \{1\}^*$ then the word $vwwxyyz$ has one border increased in depth and also the subword 10^k1 replaced by $10^{k+h}1$ where $k < k+h \leq 2k$. Thus $10^{k+h}1$ occurs twice as a subword of $vwwxyyz$ and therefore $vwwxyyz \notin L$; similarly one can see that if $w \in \{1\}^*$ and $y \in \{0\}^+$ this also causes $vwwxyyz \notin L$.

The remaining case is that both $w, y \in \{0\}^+$. Now one of them is, due to choice, a subword of 10^k1 of length h, the other one is a subword of $10^{k'}1$ of length h' for some h, h', k'. If $k' = k$ then $vwwxyyz$ contains the word $10^{h+h'+k}1$ twice (where $h+h' \leq k$ by w, y being disjoint subwords of 10^k1); if $k' \neq k$ then the only way to avoid that $vwwxyyz$ contains $10^{k+h}1$ twice is to assume that $k' = k+h$ and the occurrence of $10^{k+h}1$ in u gets pumped as well — however, then $10^{k'+h'}1$ occurs in $vwwxyyz$ twice, as $k+1 \leq k+h \leq 2k$ and $k+h+1 \leq k'+h' \leq 4k$.

So it follows by case distinction that L does not satisfy Ogden's Lemma. Thus L cannot be a context-free language. ∎

Theorem 5.24: Square-Containing Words (Ehrenfeucht and Rozenberg [23], Ross and Winklmann [64]). *The language L of the square-containing ternary words is not context-free.*

Proposition 5.25. *The language L of the square-containing ternary words satisfies Ogden's Lemma.*

Proof. This can be shown for the constant 7. Let u be any word in L and assume that at least 7 letters are marked.

Consider those splittings $vwxyz = u$ where the following conditions are met:

- v ends with the same letter a with which z starts;
- w contains at least one marked letter;
- x, y are ε.

Among all possible such splittings, choose one where w is as short as possible.

First one has to show that such a splitting exists. Let c be a marked letter such that there are at least three marked letters before and at least three marked letters after c. If there is any letter a which occurs both before and after c, then one could choose v as the letters up to some occurrence of a before c, z as the letters at and beyond some occurrence of a after c and w to be all the letters in between. If no such a exists then there is one letter which only occurs before c and two letters which only occurs after c or vice versa, say the first. Hence one letter a occurs three times marked before c and then one can split with v up to the first marked occurrence of a, w between the first and third marked occurrence of a and z at and after the third marked occurrence of a. Thus there is such a splitting and now one takes it such that w is as short as possible.

If w would contain three marked letters of the same type b then in the above the word $\tilde{v}$ ending with the first of these b, the word $\tilde{z}$ starting from the third of these b and the word $\tilde{w}$ of the letters in between with $\tilde{x}, \tilde{y}$ being ε would also appear in the list and therefore w would not be as short as possible.

Thus for each of $0, 1, 2$, there are only two marked letters of this type inside w and w contains at most six marked letters. Now $vz = vxz \in L$ as it contains aa as a subword. Furthermore $vw^\ell xy^\ell z \in L$ for $\ell > 1$, as w is not empty. $vwxyz = u \in L$ by choice of u. Thus the language L satisfies Ogden's Pumping Lemma with constant 7. ∎

A direct example not using cited results can also be constructed as follows.

Example 5.26. *Consider the language L of all words $w \in \{0,1\}^*$ such that either $w \in \{0\}^* \cup \{1\}^*$ or the difference n between the number of 0 and number of 1 in w is a cube, that is, in $\{\ldots, -64, -27, -8, -1, 0, 1, 8, 27, 64, \ldots\}$. The language L satisfies Ogden's Pumping Lemma but is not context-free.*

Proof. The language $L \cap (\{1\} \cdot \{0\}^+)$ equals to $\{10^{n^3+1} : n \in \mathbb{N}\}$. If L is context-free so must be this intersection; however, it is easy to see that this intersection violates Theorem 1.35 (b).

For the verification of Ogden's Pumping Lemma with constant 2, consider a word $u \in L$ which contains both, zeroes and ones, and consider the case that at least one letter is marked. Split the word u into $vwxyz$ such that w, y consist of two different letters (one is 0 and one is 1) and at least of one of these two letters is marked and no letter in x is marked. If letters of both types of marked, one takes that pair of marked different letters which are as near to each other as possible; if only zeroes are marked, one picks a 1 and takes for the other letter the nearest 0 which is marked; if only ones are marked, one picks a 0 anywhere in the word and choses for the other letter the nearest 1 which is marked. Then the part x between w and y picked does not contain any marked letter, as otherwise the condition to choose the "nearest marked letter" would be violated. Furthermore, v is the part before w and z is the part after y in the word u.

Now every word of the form $vw^\ell xy^\ell z$ has the same difference between the occurrences of 0 and 1 as the original word u; thus all $vw^\ell xy^\ell z$ are in L and so Ogden's Pumping Lemma is satisfied. ∎

Exercise 5.27. *Prove that the language*

$$L = \{a^h \cdot w : a \in \{0,1,2\}, w \in \{0,1,2\}^*, w \text{ is square-free and } h \in \mathbb{N}\}$$

satisfies Theorem 1.35 (b) but does not satisfy Ogden's Pumping Lemma. The fact that there are infinitely many square-free words can be used without proof; recall that a word w is square-free iff it does not have a subword of the form vv for any non-empty word v.

Exercise 5.28. *Use the Block Pumping Lemma to prove the following variant of Ogden's Lemma for regular languages: If a language L satisfies the Block Pumping Lemma with constant $k+1$ then one can, for each word u of length at least k with having at least k marked symbols, find a splitting of the word into parts x, y, z such that $u = xyz$ and $xy^*z \subseteq L$ and y contains at least 1 and at most k marked symbols.*

Example 5.29. Let L be the language generated by the grammar

$$(\{S\}, \{0,1\}, \{S \to 0S0|1S1|00|11|0|1\}, S),$$

that is, L is the language of all binary non-empty palindromes. For a grammar in Greibach Normal Form for L, one needs two additional non-terminals T, U and updates the rules as follows:

$$S \to 0ST|1SU|0T|1U|0|1, T \to 0, U \to 1.$$

Let H be the language generated by the grammar

$$(\{S\}, \{0,1\}, \{S \to SS|0S1|1S0|10|01\}, S),$$

that is, H is the language of all non-empty binary words with same number of 0 and 1. For a grammar in Greibach Normal Form for H, one needs two additional non-terminals T, U and updates the rules as follows:

$$S \to 0SU|0U|1ST|1T, T \to 0|0S, U \to 1|1S.$$

In all grammars above, the alphabet is $\{0,1\}$ and the start symbol is S.

Exercise 5.30. *Let L be the first language from Example 5.29. Find a grammar in Greibach Normal Form for $L \cap 0^*1^*0^*1^*$.*

Exercise 5.31. *Let H be the second language from Example 5.29. Find a grammar in Greibach Normal Form for $H \cap 0^*1^*0^*1^*$.*

6 Deterministic Membership Testing

For regular languages, a finite automaton can with one scan of the word decide whether the word is in the language or not. A lot of research had been dedicated to develop mechanisms for deciding membership of context-free and context-sensitive languages. For context-free languages, the algorithms use polynomial time, for context-sensitive languages, they can do in polynomial space and it is conjectured that it is in some cases impossible to get polynomial time. The conjecture is equivalent to the conjecture that the complexity classes P and PSPACE (polynomial time and polynomial space, respectively) are different. This difference is implied by the conjecture that P is different from NP; the latter is believed by 83% of the people in complexity theory who participated in a recent poll by Bill Gasarch [28].

Cocke [17], Kasami [44] and Younger [77] developed independently of each other an algorithm which solves the membership of a word in a given context-free grammar in time $O(n^3)$. For this algorithm, the grammar is fixed and its size is considered to be constant; if one factors the size of the grammar in, then the algorithm is $O(n^3 \cdot m)$ where m is the size of the grammar (number of rules).

Algorithm 6.1: Cocke, Kasami and Young's Parser [17, 44, 77]**.** Let a context-free grammar (N, Σ, P, S) be given in Chomsky Normal Form and let w be a non-empty input word. Let $1, 2, \ldots, n$ denote the positions of the symbols in w, so $w = a_1 a_2 \ldots a_n$. Now one defines variables $E_{i,j}$ with $i < j$, each of them taking a set of non-terminals, as follows:

1. Initialisation: For all k,

$$E_{k,k} = \{A \in N : A \to a_k \text{ is a rule}\}.$$

2. Loop: Go through all pairs (i, j) such that they are processed in increasing order of the difference $j - i$ and let

$$E_{i,j} = \{A : \exists \text{ rule } A \to BC \, \exists k \, [i \leq k < j \text{ and } B \in E_{i,k} \text{ and } C \in E_{k+1,j}]\}.$$

3. Decision: w is generated by the grammar iff $S \in E_{1,n}$.

To see that the run-time is $O(n^3 \cdot m)$, note that the initialisation takes time $O(n \cdot m)$ and that in the loop, one has to fill $O(n^2)$ entries in the right order. Here each entry is a vector of up to m bits to represent which of the non-terminals are represented; initially they are 0. Then for each rule $A \to BC$ and each k with $i \leq k < j$, one checks whether the entry for B in the vector for $E_{i,k}$ and the entry for C in the vector for $E_{k+1,j}$ are 1; if so, one adjusts the entry for A in the vector of $E_{i,j}$ to 1. This loop runs over $O(n \cdot m)$ entries with n being a bound on the number of values k can take and m being the number of rules, thus each of the variables $E_{i,j}$ is filled in time $O(n \cdot m)$ and the overall time complexity of the loop is $O(n^3 \cdot m)$. The decision is $O(1)$. Thus the overall time complexity is $O(n^3 \cdot m)$.

Example 6.2. Consider the grammar $(\{S, T, U\}, \{0, 1\}, \{S \to SS|TU|UT, U \to 0|US|SU, T \to 1|TS|ST\}, S)$ and the word 0011. Now one can compute the entries of the $E_{i,j}$ as follows:

$$\begin{array}{ccccccc} & & & E_{1,4} = \{S\} & & & \\ & & E_{1,3} = \{U\} & & E_{2,4} = \{T\} & & \\ & E_{1,2} = \emptyset & & E_{2,3} = \{S\} & & E_{3,4} = \emptyset & \\ E_{1,1} = \{U\} & & E_{2,2} = \{U\} & & E_{3,3} = \{T\} & & E_{4,4} = \{T\} \\ 0 & & 0 & & 1 & & 1 \end{array}$$

As $S \in E_{1,4}$, the word 0011 is in the language. Now consider the word 0111.

$$\begin{array}{ccccccc} & & & E_{1,4} = \emptyset & & & \\ & & E_{1,3} = \{T\} & & E_{2,4} = \emptyset & & \\ & E_{1,2} = \{S\} & & E_{2,3} = \emptyset & & E_{3,4} = \emptyset & \\ E_{1,1} = \{U\} & & E_{2,2} = \{T\} & & E_{3,3} = \{T\} & & E_{4,4} = \{T\} \\ 0 & & 1 & & 1 & & 1 \end{array}$$

As $S \notin E_{1,4}$, the word 0111 is not in the language.

Quiz 6.3. *Let the grammar be the same as in the previous example. Make the table for the word* 1001.

Exercise 6.4. *Consider the grammar* $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ *with* P *consisting of the rules* $S \to TT$, $T \to UU|VV|WW$, $U \to VW|WV|VV|WW$, $V \to 0$, $W \to 1$. *Make the entries of the Algorithm of Cocke, Kasami and Younger for the words* 0011, 1100 *and* 0101.

Exercise 6.5. *Consider the grammar* $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ *with* P *consisting of the rules* $S \to ST|0|1$, $T \to TU|1$, $U \to UV|0$, $V \to VW|1$, $W \to 0$. *Make the entries of the Algorithm of Cocke, Kasami and Younger for the word* 001101.

Description 6.6: Linear Grammars. A linear grammar is a grammar where each derivation has in each step at most one non-terminal. Thus every rule is either of the form $A \to u$ or $A \to vBw$ for non-terminals A, B and words u, v, w over the terminal alphabet. For parsing purposes, it might be sufficient to make the algorithm for dealing with non-empty words and so one assumes that ε is not in the language.

As in the case of the Chomsky Normal Form, one can put every linear language in a normal form where all rules are either of the form $A \to c$ or $A \to cB$ or $A \to Bc$ for non-terminals A, B and terminals c. This permits to adjust the algorithm of Cocke, Kasami and Younger to the case of linear grammars where it runs in time $O(n^2 \cdot m)$, where n is the length of the input word and m is the number of rules.

Algorithm 6.7: Parsing of Linear Grammars. Let a linear grammar (N, Σ, P, S) be given in the normal form from Description 6.6 and let w be a non-empty input word. Let $1, 2, \ldots, n$ denote the positions of the symbols in w, so $w = a_1 a_2 \ldots a_n$. Now one defines variables $E_{i,j}$ with $i < j$, each of them taking a set of non-terminals, as follows:

1. Initialisation: For all k,

$$E_{k,k} = \{A \in N : A \to a_k \text{ is a rule}\}.$$

2. Loop: Go through all pairs (i, j) such that they are processed in increasing order of the difference $j - i$ and let

$$\begin{aligned} E_{i,j} = \{A :\ & \exists \text{ rule } A \to Bc \quad [B \in E_{i,j-1} \text{ and } c = a_j] \text{ or} \\ & \exists \text{ rule } A \to cB \quad [c = a_i \text{ and } B \in E_{i+1,j}]\}. \end{aligned}$$

3. Decision: w is generated by the grammar iff $S \in E_{1,n}$.

To see that the run-time is $O(n^2 \cdot m)$, note that the initialisation takes time $O(n \cdot m)$ and that in the loop, one has to fill $O(n^2)$ entries in the right order. Here each entry is a vector of up to m bits to represent which of the non-terminals are represented; initially the bits are 0. Then for each rule, if the rule is $A \to Bc$ one checks whether $B \in E_{i,j-1}$ and $c = a_j$ and if the rule is $A \to cB$ one checks whether $B \in E_{i+1,j}$ and $c = a_i$. If the check is positive, one adjusts the entry for A in the vector of $E_{i,j}$ to 1. This loop runs over $O(m)$ entries with m being the number of rules, thus each of the variables $E_{i,j}$ is filled in time $O(m)$ and the overall time complexity of the loop is $O(n^2 \cdot m)$. The decision is $O(1)$. Thus the overall time complexity is $O(n^2 \cdot m)$.

Example 6.8. Consider the grammar

$$(\{S, T, U\}, \{0, 1\}, \{S \to 0|1|0T|1U, T \to S0|0, U \to S1|1\}, S)$$

which is a linear grammar generating all non-empty binary palindromes. Then one gets the following table for processing the word 0110:

$$
\begin{array}{ccccccc}
 & & & E_{1,4} = \{S\} & & & \\
 & & E_{1,3} = \emptyset & & E_{2,4} = \{T\} & & \\
 & E_{1,2} = \{U\} & & E_{2,3} = \{S,U\} & & E_{3,4} = \{T\} & \\
E_{1,1} = \{S,T\} & & E_{2,2} = \{S,U\} & & E_{3,3} = \{S,U\} & & E_{4,4} = \{S,T\} \\
0 & & 1 & & 1 & & 0
\end{array}
$$

As $S \in E_{1,4}$, the word is accepted. Indeed, 0110 is a palindrome. For processing the word 1110, one gets the following table:

$$
\begin{array}{ccccccc}
 & & & E_{1,4} = \{T\} & & & \\
 & & E_{1,3} = \{S,U\} & & E_{2,4} = \{T\} & & \\
 & E_{1,2} = \{S,U\} & & E_{2,3} = \{S,U\} & & E_{3,4} = \{T\} & \\
E_{1,1} = \{S,U\} & & E_{2,2} = \{S,U\} & & E_{3,3} = \{S,U\} & & E_{4,4} = \{S,T\} \\
1 & & 1 & & 1 & & 0
\end{array}
$$

As $S \notin E_{1,4}$, the word is rejected. It is easy to see that 1110 is not a palindrome and that the algorithm is also correct in this case.

Exercise 6.9. *For the grammar from Example 6.8, construct the table for the algorithm on the word* 0110110.

Exercise 6.10. *Consider the following linear grammar:*

$$(\{S,T,U\}, \{0,1\}, \{S \to 0T|T0|0U|U0, T \to 0T00|1, U \to 00U0|1\}, S).$$

Convert the grammar into the normal form from Description 6.6 and construct then the table of the algorithm for the word 00100.

Exercise 6.11. *Which two of the following languages are linear? Provide linear grammars for these two languages:*

- $L = \{0^n 1^m 2^k : n + k = m\}$;
- $H = \{0^n 1^m 2^k : n + m = k\}$;
- $K = \{0^n 1^m 2^k : n \neq m \text{ or } m \neq k\}$.

Algorithm 6.12: Kleene Star of Linear Grammar. Let L be a linear grammar. Then there is an $O(n^2)$ algorithm which can check whether a word w of length n is in L^*. Let $w = a_1 a_2 \ldots a_n$ be the input word and n be its length.

First Part: Compute for each i, j with $1 \leq i \leq j \leq n$ the set $E_{i,j}$ of all non-terminals which generate $a_i a_{i+1} \ldots a_j$.

Initialise Loop for Kleene Star: Let $F_0 = 1$.

Loop for Kleene Star: For $m = 1, 2, \ldots, n$ Do
Begin If there is a $k < m$ with $S \in E_{k+1,m}$ and $F_k = 1$
Then let $F_m = 1$ Else let $F_m = 0$ End.

Decision: $w \in L^*$ iff $F_n = 1$.

The first part is in $O(n^2)$ as the language is linear, see Algorithm 6.7. The Loop for Kleene Star can be implemented as a double loop on the variables m and k and runs in $O(n^2)$. The decision is afterwards reached by just checking one variable. Thus the overall complexity is $O(n^2)$. For correctness, one just has to prove by induction that $F_m = 1$ iff $a_1 \ldots a_m$ is in L^*. Note that $F_0 = 1$ as the empty word is in L^* and the inductive equation is that

$$a_1 \ldots a_m \in L^* \Leftrightarrow \exists k < m \, [a_1 \ldots a_k \in L^* \text{ and } a_{k+1} \ldots a_m \in L]$$

which is implemented in the search; note that k can be 0 in the case that $a_1 \ldots a_m \in L$.

Exercise 6.13. *Construct a quadratic time algorithm which checks whether a word u is in $H \cdot K \cdot L$ where H, K, L are linear languages. The subalgorithms to make the entries which of the subwords of u are in H, K, L can be taken over from Algorithm 6.7.*

Exercise 6.14. *Construct a quadratic time algorithm which checks whether a word u is in $(L \cap H)^* \cdot K$ where H, K, L are linear languages. The subalgorithms to make the entries which of the subwords of u are in H, K, L can be taken over from Algorithm 6.7.*

In the following exercise, one uses as the base case of regular expressions not finite lists of words but arbitrary context-free languages. An example is to take two context-free languages L_1, L_2 and then to consider expressions like

$$((L_1 \cap L_2)^* \cdot L_1 \cdot L_2 \cdot (L_1 \cap L_2)^*)^+$$

and then the question is on how difficult the membership test for such a language is. The main task of the exercise is to show that each such language has an $O(n^3)$ parsing algorithm.

Exercise 6.15. *Let the base case of expressions in this exercise be context-free languages and combine those by concatenation, union, intersection, set-difference, Kleene Star and Kleene Plus. Consider regular expression with context-free languages as primitive parts in the language which are combined by the given connectives. Now describe by structural induction on how to construct an $O(n^3)$ decision procedure for* *languages of this type.*

The key idea is that whenever one combines one or two languages with concatenation, intersection, union, set difference, Kleene Plus or Kleene star, one can from algorithms which provide for any given subword $a_i \ldots a_j$ of the input word $a_1 \ldots a_n$ a value $E_{i,j}, \tilde{E}_{i,j} \in \{0,1\}$ denoting whether the subword is in or not in the language L or $\tilde{L}$, respectively, create an algorithm which does the same for $L \cap \tilde{L}$, $L \cup \tilde{L}$, $L - \tilde{L}$, $L \cdot \tilde{L}$, L^* and L^+. Show that the corresponding computations of the new entries are always in $O(n^3)$.

For context-sensitive languages, only a quadratic-space algorithm is known due to a construction by Savitch [67]. Note that when there are at most k^n different words of non-terminals and terminals up to length n then the length of the shortest derivation is also at most k^n. Furthermore, one can easily check whether in a derivation $v \Rightarrow w$ can be done in one step by a given grammar.

Algorithm 6.16. Let a context-sensitive grammar (N, Σ, P, S) for a language L be given and let $k = |N| + |\Sigma| + 1$. For some input $n > 0$ and $w \in \Sigma^n$, the following algorithm checks in space $O(n^2)$ whether w can be derived from S; note that each call of the subroutine needs to archive the local variables u, v, t on the stack and this uses $O(n)$ space as the words u, v have up to length n with respect to the finite alphabet $N \cup \Sigma$ and t is a number below k^n which can be written down with $O(n)$ digits.

Recursive Call: Function $Check(u, v, t)$
Begin If $u = v$ or $u \Rightarrow v$ Then Return(1);
If $t \leq 1$ and $u \neq v$ and $u \not\Rightarrow v$ Then Return(0);
Let $t' = Floor(\frac{t+1}{2})$; Let $r' = 0$;
For all $u' \in (N \cup \Sigma)^*$ with $|u| \leq |u'| \leq |v|$ Do
Begin If $Check(u, u', t') = 1$ and $Check(u', v, t') = 1$ Then $r' = 1$ End;
Return(r') End;

Decision: If $Check(S, w, k^n) = 1$ Then $w \in L$ Else $w \notin L$.

For the verification, let k' be a number with $2^{k'} \geq k$. Then one can see, by easy induction, that Check is first called with $2^{k' \cdot n}$ or less and then at each iteration of the call, the value of t' is half of the value of t so that the number of iterated calls is at most $k' \cdot n$. Thus the overall space to archive the stacks used is at most $(k' \cdot n) \cdot 4 \cdot k' \cdot n$ where $k' \cdot n$ is the number of nested calls, 4 is the number of variables to be archived (u, v, u', t') and $k' \cdot n$ is the space needed (in bits) to archive these numbers. Some minor space might also be needed for local processing within the loop.

For the verification of the correctness of $Check(u, v, t)$, note that when v is derived from u, all intermediate words are at least as long as u and at most as long as v, thus the intermediate word u' in the middle is, if v can derived from u within t steps, within the search space. As one can process the search-space in length-lexicographic order, it is enough to memorise u' plus r' plus t' plus the outcome of the first call $Check(u, u', t')$ when doing the second call $Check(u', v, t')$. So the local space of an instance of $Check$ can indeed be estimated with $O(n)$. Furthermore, when $t > 1$, there must be an intermediate u' which is reached in the middle of the derivation from u to v, and one can estimate the time t' from u to u' as well as from u' to u in both cases with $Floor(\frac{t+1}{2})$.

The runtime of the algorithm is $O(k^{2n^2})$. One can easily see that one instance of $Check(u, v, t)$ without counting the subroutines runs in time $O(k^n)$, furthermore, each $Check(u, v, t)$ calls $2 \cdot k^n$ times a subroutine with parameter $t/2$. The number of nesting of the calls is $\log(k^n) = \log(k) \cdot n$ which gives $O((2 \cdot k^n)^{\log(k) \cdot n})$ which can be bounded by $O((2k)^{\log(k) \cdot n^2})$. Furthermore, as every call itself is $O(k^n)$ in runtime, so the overall runtime is $O((2k)^{\log(k) \cdot n^2 + n})$ which can be simplified to an upper bound of $O(c^{n^2})$ for any constant $c > (2k)^{\log(k)}$. Up to today no subexponential algorithms are known for this problem.

Example 6.17. There is a context-sensitive grammar where for each length n there is a word of n symbols which cannot be derived in less than 2^n steps. This bound is only to be true for the grammar constructed, other grammars for the same language can have better bounds.

The grammar $(\{S, U, V, W\}, \{0, 1\}, P, S)$ simulates binary counting and has the following rules in P:

$S \to 0S|U$, $U \to V|0$, $0V \to 1U$, $V \to 1$, $1V \to WU$, $1W \to W0$, $0W \to 10$.

The binary counter starts with generating $n-1$ digits 0 and then deriving from S to U. U stands for the last digit 0, V stands for last digit 1, W stands for a digit 0 still having a carry bit to pass on. Deriving a binary number k needs at least k steps. So deriving 1^n representing $2^n - 1$ in binary requires $2^n - 1$ counting steps where every fourth counting step requires a follow-up with some carry, so that one can even show that for 1^n more than 2^n derivation steps are needed.

Exercise 6.18. *Give a proof that there are k^n or less words of length up to n over the alphabet $\Sigma \cup N$ with $k-1$ symbols.*

Exercise 6.19. *Modify Savitch's Algorithm such that it computes the length of the shortest derivation of a word w in the context-sensitive grammar, provided that such derivation exists. If it does not exist, the algorithm should return the special value ∞.*

Exercise 6.20. *Consider the following algorithm:*

Recursive Call: Function $Check(u,w,t)$

Begin If $u=w$ or $u \Rightarrow w$ Then Return(1);

If $t \leq 1$ and $u \neq v$ and $u \not\Rightarrow w$ Then Return(0);

Let $r' = 0$; For all $v \in (N \cup \Sigma)^*$ with $u \Rightarrow v$ and $|v| \leq |w|$ Do

Begin If $Check(v,w,t-1) = 1$ Then $r' = 1$ End;

Return(r') End;

Decision: If $Check(S,w,k^n) = 1$ Then $w \in L$ Else $w \notin L$.

Analyse the time and space complexity of this algorithm. Note that there is a polynomial time algorithm which returns to given u,w the list of all v with $u \Rightarrow v$ and $|v| \leq |w|$.

Definition 6.21: Growing CTS by Dahlhaus and Warmuth [18]**.** *A grammar (N,Σ,P,S) is growing iff $|l| < |r|$ for all rules $l \to r$ in the grammar.*

So growing grammars are context-sensitive by the corresponding characterisation, thus they are also called "growing context-sensitive grammars". It is clear that their membership problem is in polynomial space. An important result is that this problem is also in polynomial time.

Theorem 6.22: Growing CTS Membership is in P (Dahlhaus and Warmuth [18]). *Given a growing context-sensitive grammar there is a polynomial time algorithm which decides membership of the language generated by this growing grammar.*

In this result, polynomial time means here only with respect to the words in the language, the dependence on the size of the grammar is not polynomial time. So if one asks the uniform decision problem for an input consisting of a pair of a grammar and a word, no polynomial time algorithm is known for this problem. As the problem is NP-complete, the algorithm is unlikely to exist.

Example 6.23. Consider the grammar

$$(\{S,T,U\},\{0,1\},\{S \to 011|T11, T \to T0U|00U, U0 \to 0UU, U1 \to 111\}, S)$$

which is growing. This grammar has derivations like $S \Rightarrow T11 \Rightarrow 00U11 \Rightarrow 001111$ and $S \Rightarrow 0T11 \Rightarrow T0U11 \Rightarrow 0U0U11 \Rightarrow 00U01111 \Rightarrow 000UU1111 \Rightarrow 000U111111 \Rightarrow 00011111111$. The language of the grammar is

$$\{0^n 1^{2^n} : n > 0\} = \{011, 001111, 0^3 1^8, 0^4 1^{16}, 0^5 1^{32}, \ldots\}$$

and not context-free. The latter can be seen as infinite languages satisfying the pumping lemma can only have constant gaps, that is, there is a maximum constant

c such that for some t there are no words of length $t, t+1, \ldots, t+c$ in the language. However, the gaps of this language are growing, each sequence $n+2^n+1, n+2^n+2, \ldots, n+2^{n+1}$ is a gap.

Exercise 6.24. *Show that every context-free language is the union of a language generated by a growing grammar and a language containing only words up to length 1.*

Exercise 6.25. *Modify the proof of Theorem 4.11 to prove that every recursively enumerable language, that is, every language generated by some grammar is the homomorphic image of a language generated by a growing context-sensitive grammar.*

Exercise 6.26. *Construct a growing grammar for the language* $\{1^{2^n}0^{2n}1^{2^n} : n > 0\}$ *which is the "palindromisation" of the language from Example 6.23.*

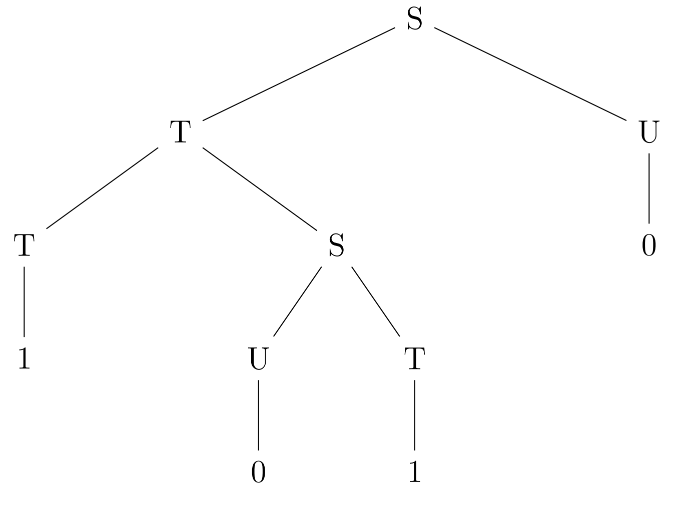
7 Non-Deterministic Membership Testing

For finite automata, non-deterministic automata and deterministic automata do not vary in speed to process data, only in the amount of states needed for a given regular language. For membership testing of context-free languages, there is, up to current knowledge, a significant difference in speed. Non-deterministic algorithms, so called pushdown automata, can operate with speed $O(n)$ on the words while deterministic algorithms like the one of Cocke, Kasami and Younger need $O(n^3)$ or, with some improvements by exploiting fast matrix multiplication algorithms, about $O(n^{2.3728639})$.

Description 7.1: Push Down Automaton. The basic idea for the linear time algorithm to check non-deterministically membership in a context-free language is that, for a grammar in Chomsky Normal Form, a word of length n can be derived in $2n-1$ steps, $n-1$ applications of rules which convert one non-terminal into two, n applications of rules which convert a non-terminal into a terminal. A second idea used is to go through the derivation tree and to do the left-most rule which can be applied. Here an example with the usual grammar

$$(\{S,T,U\},\{0,1\},\{S \to SS|TU|UT, U \to 0|US|SU, T \to 1|TS|ST\},S).$$

and the derivation tree for the derivation $S \Rightarrow^* 1010$:



Now the left-most derivation according to this tree is $S \Rightarrow TU \Rightarrow TSU \Rightarrow 1SU \Rightarrow 1UTU \Rightarrow 10TU \Rightarrow 101U \Rightarrow 1010$. Note that in each step the left-most non-terminal is replaced by something else using the corresponding rule. The idea of the algorithm is now to split the data of the derivation into two parts:

- The sequence of the so far generated or compared terminals;

- The sequence of the current non-terminals in the memory.

The sequence of non-terminals behave like a stack: The first one is always processed and then the new non-terminals, if any, are pushed to the front of the stack. The terminals are, whenever generated, compared with the target word; alternatively, one can therefore also read the terminals symbol by symbol from the source and whenever one processes a rule of the form $A \to a$ one compares this a with the current terminal: if they agree then one goes on with the derivation else one discards the work done so far. Such a concept is called a pushdown automaton — where non-terminal symbols are pushed down into the stack of the non-terminals or pulled out when the current non-terminal has just been converted into a terminal. The pushdown automaton would therefore operate as follows:

Start: The symbol S is on the stack.

Loop: While there are symbols on the stack Do Begin

Step 1: Pull a symbol A from the top of the stack;

Step 2: Select non-deterministically a rule $A \to w$ from the set of rules;

Step 3a: If the rule is $A \to BC$ Then push BC onto the stack so that B becomes the topmost symbol and continue the next iteration of the loop;

Step 3b: If the rule is $A \to a$ Then Read the next symbol b from the input;
If there is no next symbol b on the input or if $b \neq a$ then abort the computation else continue the next iteration of the loop End End;

Decision: If all symbols from the input have been read and the computation has not yet been aborted Then accept Else reject.

For a more formal treatment, one also allows states in the push down automaton.

Definition 7.2: Pushdown Automaton. A pushdown automaton consists of a tuple $(Q, \Sigma, N, \delta, s, S, F)$ where Q is a set of states with s being the start state and F being the set of final states, Σ is the alphabet used by the target language, δ is the transition function and N is the set of stack symbols with the start symbol S being on the stack.

In each cycle, the push down automaton currently in state p pulls the top element A from the stack and selects a rule from $\delta(p, A, v)$ which consists of a pair (p, w) where $v \in \Sigma^*$ and $w \in N^*$; if v agrees with the next input symbols to be processed (this is void if $v = \varepsilon$) then the automaton advances on the input by these symbols and pushes

w onto the stack and takes the new state q.

There are two ways to define when a pushdown automaton accepts: Acceptance by state means that the pushdown automaton accepts iff there is a run starting with S on the stack that goes through the cycles until the automaton has processed all input and is in an accepting state. Acceptance by empty stack means that the pushdown automaton accepts iff there is a run starting with S on the stack that goes through the cycles until the automaton has processed all input and is in an accepting state and the stack is empty. Note that the automaton gets stuck if it has not yet read all inputs but there is no symbol left on the stack; such a run is considered as rejecting and cannot count as an accepting run.

A common convention is that the word v of the input to be parsed in a cycle always consists of either one symbol or zero symbols.

Example 7.3. The pushdown automaton from the beginning of this chapter has the following description:

- $Q = \{s\}$ and $F = \{s\}$;
- $\Sigma = \{0, 1\}$;
- $N = \{S, T, U\}$ with start symbol S;
- $\delta(s, \varepsilon, S) = \{(s, SS), (s, TU), (s, UT)\}$;
  $\delta(s, \varepsilon, T) = \{(s, TS), (s, ST)\}$;
  $\delta(s, \varepsilon, U) = \{(s, US), (s, SU)\}$;
  $\delta(s, 0, U) = \{(s, \varepsilon)\}$;
  $\delta(s, 1, T) = \{(s, \varepsilon)\}$;
  $\delta(s, v, A) = \emptyset$ for all other choices of (v, A).

Here a sample processing of the word 001110:

input processed	start	–	0	–	–	0	1	–	1	–	1	0
new stack	S	UT	T	ST	UTT	TT	T	TS	S	TU	U	ε

One can generalise this idea to an algorithm which works for any context-free language given by a grammar in Chomsky Normal Form.

Algorithm 7.4. Let (N, Σ, P, S) be a grammar in Chomsky Normal Form generating a language L. Then one can construct a pushdown automaton recognising the same language L as follows:

- $Q = \{s\}$ and $F = \{s\}$;

- Σ and N are taken over from the grammar; furthermore, S is again the start symbol;
- For every non-terminal $A \in N$, one defines that $\delta(s, \varepsilon, A) = \{(s, BC) : A \to BC \text{ is in } P\} \cup \{(s, \varepsilon) : S \to \varepsilon \text{ is in } P \text{ and } A = S\}$, for $a \in \Sigma$, if $A \to a$ is in P then $\delta(s, a, A) = \{(s, \varepsilon)\}$ else $\delta(s, a, A) = \emptyset$.

This algorithm was applied in Example 7.3.

Verification. One shows by induction over the derivation length that the automaton can have the stack w after processing input v iff $v \in \Sigma^*$, $w \in N^*$ and $S \Rightarrow^* vw$. Here the derivation length is the number of steps of the pushdown automaton or the number of steps in the derivation.

It is clear that S can derive only S in zero steps and that the pushdown automaton, similarly, has S in the stack and no input processed after zero steps. Furthermore, one can see that ε is derived in the grammar in exactly one step iff $S \to \varepsilon$ is in P iff the pushdown automaton has $\delta(s, \varepsilon, S) = \{(s, \varepsilon)\}$ iff the pushdown automaton can go in one step into the situation where no input has been processed so far and $w = \varepsilon$.

Now consider that the equivalence holds for k steps, it has to be shown that it also holds for $k + 1$ steps.

Now assume that the pushdown automaton has processed v in $k+1$ steps and has the stack w and assume that $vw \neq \varepsilon$, as that case has already been covered. Let $\tilde{v}$ and $\tilde{w}$ be the processed input in the first k steps and $\tilde{w}$ be the stack after k steps on the way to v and w. There are two cases:

First, $\tilde{w} = Aw$ and $\tilde{v}a = v$. Then $\delta(s, a, A) = \{(s, \varepsilon)\}$ and the rule $A \to a$ is in P. Furthermore, $S \Rightarrow^* \tilde{v}A\tilde{w}$ in k steps. Now one can apply the rule $A \to a$ and obtains that $S \Rightarrow^* \tilde{v}a\tilde{w} = vw$ in $k + 1$ steps.

Second, $w = BC\tilde{w}$ and the pushdown automaton had processed v already at step k and had the stack $A\tilde{w}$. Then the pushdown automaton satisfies $(s, BC) \in \delta(s, \varepsilon, A)$ and $A \to BC$ is a rule. Furthermore, $S \Rightarrow^* vA\tilde{w}$ in k steps in the grammar and now applying $A \to BC$ gives $S \Rightarrow^* vBC\tilde{w}$ in $k + 1$ steps, the righthand side is equal to vw.

Furthermore, for the other way round, assume that $S \Rightarrow vw$ in $k + 1$ steps in a left-most derivation. Again assume that $vw \neq \varepsilon$.

First, if the last rule was $A \to a$, then $S \to \tilde{v}Aw$ in k steps and $v = \tilde{v}a$. By induction hypothesis, the pushdown automaton can process $\tilde{v}$ in k processing steps having a memory Aw and then in the next processing step read a and use up A, as $\delta(s, A, a) = \{(s, \varepsilon)\}$ so that the pushdown automaton has read v and produced the stack of w in $k + 1$ steps.

Second, if the last rule applied was $A \to BC$ then $S \to vA\tilde{w}$ in k steps with $w = BC\tilde{w}$ and the pushdown automaton can process v and having step $A\tilde{w}$ after k

steps. Furthermore, $(s, BC) \in \delta(s, \varepsilon, A)$ and therefore the pushdown automaton can have the stack $BC\tilde{w} = w$ after $k+1$ steps with the input processed still being the same v.

This induction shows that $S \Rightarrow^* vw$ with $v \in \Sigma^*$ and $w \in N^*$ iff there is a run of the pushdown automaton which starts on stack being S and which has, after reading input v and doing some processing the stack w. This equivalence is in particular true if $w = \varepsilon$ and therefore the words generated by the grammar are the same as those accepted by the pushdown automaton in some run.

Exercise 7.5. *Construct a pushdown automaton accepting by empty stack for the language* $\{0^n 1^m 2^k : n + m = k + 1\}$.

Exercise 7.6. *Construct a pushdown automaton accepting by empty stack for the language* $\{0^n 1^m 2^k : n + m < k\}$.

Exercise 7.7. *Construct a pushdown automaton accepting by empty stack for the language* $\{0^n 1^m 2^k : n \neq m \text{ and } k > 0\}$.

The next algorithm describes how to generate a pushdown automaton accepting by state from a given context-free grammar in Chomsky Normal Form; the verification is similar and therefore only sketched.

Algorithm 7.8. Assume that (N, Σ, P, S) is a context-free grammar in Chomsky Normal Form. Then the following pushdown automaton recognises L with the acceptance method by state. The pushdown automaton is $(\{s, t\}, \Sigma, N \cup N', \delta, s, S', \{t\})$ where $N' = \{A' : A \in N\}$ is a "primed copy" of N. The primed version of a non-terminal is always the last non-terminal in the stack. For every non-terminal $A \in N$ and the corresponding $A' \in N'$, one defines the following transition function:

> $\delta(s, \varepsilon, A) = \{(s, BC) : A \to BC \text{ is a rule in } P\}$;
> $\delta(s, \varepsilon, A') = \{(s, BC') : A \to BC \text{ is a rule in } P\} \cup \{(t, \varepsilon) : A' = S'$ and $S \to \varepsilon$ is a rule in $P\}$;
> for all terminals a, if the rule $A \to a$ is in P then $\delta(s, a, A) = \{(s, \varepsilon)\}$ and $\delta(s, a, A') = \{(t, \varepsilon)\}$ else $\delta(s, a, A) = \emptyset$ and $\delta(s, a, A') = \emptyset$;
> $\delta(t, v, A), \delta(t, v, A')$ are $\emptyset$ for all v.

Similar as in the algorithm before, one can show the following: $S \Rightarrow^* vwA$ with $v \in \Sigma^*$, $w \in N^*$ and $A \in N$ iff the pushdown automaton can, on input v reach the stack content wA' and is in state s. Furthermore, the pushdown automaton can process input v and reach empty stack iff it can process v and reach the state t — the reason is that for reaching state t it has to transform the last stack symbol of the form A' into ε and this transformation always leads from s to t.

Exercise 7.9. *Construct a pushdown automaton accepting by state for the language* $\{0^n 1^m 2^k : n + m > k\}$.

Exercise 7.10. *Construct a pushdown automaton accepting by state for the language* $\{0^n 1^m 2^k : n \neq k$ *or* $m \neq k\}$.

Exercise 7.11. *Construct a pushdown automaton accepting by state for the language* $\{w \in \{0,1\}^* : w$ *is not a palindrome*$\}$.

Theorem 7.12. *If* L *can be recognised by a pushdown automaton accepting by final state then it can also be recognised by a pushdown automaton accepting by empty stack.*

Proof. Given a pushdown automaton $(Q, \Sigma, N, \delta, s, S, F)$ for L, one constructs a new automaton $(Q \cup \{t\}, N, \delta', s, S, F \cup \{t\})$ as follows (where t is a new state outside Q):

- For all $q \in Q$, $A \in N$ and v, $\delta'(q, v, A) = \delta(q, v, A) \cup \{(t, \varepsilon) : v = \varepsilon$ and $q \in F\}$;
- For all $A \in N$ and $v \neq \varepsilon$, $\delta'(t, \varepsilon, A) = \{(t, \varepsilon)\}$ and $\delta'(t, v, A) = \emptyset$.

So whenever the original pushdown automaton is in an accepting state, it can opt to remove all remaining non-terminals in the stack by transiting to t; once it transits to t, during this transit and afterwards, it does not process any input but only removes the symbols from the stack. Thus the new pushdown automaton can accept a word v by empty stack iff the old pushdown automaton can accept that word v by final state. ∎

Furthermore, one can show that whenever a language L is recognised by a pushdown automaton using the empty stack acceptance condition then it is generated by context-free grammar.

Algorithm 7.13: Pushdown Automaton to Grammar. Given a pushdown automaton $(Q, \Sigma, N, \delta, s, S, F)$ for L which accepts by empty stack, let the new grammar $((Q \times N \times Q) \cup \{S'\}, \Sigma, P, S')$ be defined by putting the following rules into P:

- For all $p \in F$, put all rules $S' \to (s, S, p)$ into P;
- For each $q, r \in Q$, $A \in N$, $v \in \Sigma^*$ and $(p_1, w) \in \delta(q, v, A)$ with $w = B_1 B_2 \ldots B_n$ with $n > 0$ and for all $p_2, \ldots, p_n \in Q$, put the rule

  $$(q, A, r) \to v(p_1, B_1, p_2)(p_2, B_2, p_3) \ldots (p_n, B_n, r)$$

  into P;

- For each $q \in Q$, $A \in N$, $v \in \Sigma^*$ and $(p, \varepsilon) \in \delta(q, v, A)$, put the rule $(q, A, p) \to v$ into P.

The idea of the verification is that a non-terminal (p, A, q) generates a word v iff the pushdown automaton can on state p process the input v with exactly using up the symbol A in some steps and ending up in state q afterwards with the stack symbols behind A being untouched. The construction itself is similar to the construction which looks at the intersection of a context-free language with a regular language and the verification is done correspondingly.

Example 7.14. Consider the following pushdown-automaton.

- $Q = \{s, t\}$, $F = \{t\}$, start state is s;
- $\Sigma = \{0, 1\}$;
- $N = \{S, U, T\}$, start symbol is S;
- $\delta(s, 0, S) = \{(s, SU), (t, U), (t, \varepsilon)\}$;
  $\delta(s, 1, S) = \{(s, ST), (t, T), (t, \varepsilon)\}$;
  $\delta(t, 0, U) = \{(t, \varepsilon)\}$; $\delta(t, 1, U) = \emptyset$;
  $\delta(t, 1, T) = \{(t, \varepsilon)\}$; $\delta(t, 0, T) = \emptyset$;
  $\delta(q, \varepsilon, A) = \emptyset$ for all $q \in Q$ and $A \in N$.

Now, one can obtain the following context-free grammar for the language:

- Set of non-terminals is $\{S'\} \cup Q \times N \times Q$ with start symbol S';
- Set of terminals is $\{0, 1\}$;
- $S' \to (s, S, t)$;
  $(s, S, t) \to 0(s, S, t)(t, U, t)|0(t, U, t)|0|1(s, S, t)(t, T, t)|1(t, T, t)|1$;
  $(t, U, t) \to 0$;
  $(t, T, t) \to 1$;
- Start symbol S'.

Unnecessary rules are omitted in this example; the set of non-terminals could just be $\{S', (s, S, t), (t, T, t), (t, U, t)\}$.

If one does not use U, T as place-holders for $0, 1$ and identifies $S', (s, S, t)$ to S, then one can get the following optimised grammar: The unique non-terminal is the start symbol S, the set of terminals is $\{0, 1\}$, the rules are $S \to 0S0|00|0|1S1|11|1$. Thus the pushdown automaton and the corresponding grammar just recognise the set of non-empty binary palindromes.

Greibach [32] established a normal form which allows to construct pushdown automata which can check the membership of a word with processing one input symbol in each step. The automaton accepts words by state.

Algorithm 7.15: Pushdown Automaton reading Input in Each Cycle. Let (N, Σ, P, S) be a grammar in Greibach Normal Form. The pushdown automaton uses the idea of primed symbols to remark the end of the stack. It is constructed as follows:

- The set of states is $\{s, t, u\}$ and if ε is in the language then $\{s, u\}$ are accepting else only $\{u\}$ is the set of accepting states; s is the start state.
- Let $N' = \{A, A' : A \in N\}$ and S' be the start symbol.
- The terminal alphabet is Σ as for the grammar.
- For all symbols $a \in \Sigma$ and $A \in N$,
  $\delta(s, a, S') = \{(t, B_1 B_2 \ldots B_n') : S \to a B_1 B_2 \ldots B_n$ is a rule in P with $n > 0\} \cup \{(u, \varepsilon) : S \to a$ is a rule in $P\}$;
  $\delta(t, a, A) = \{(t, B_1 B_2 \ldots B_n) : S \to a B_1 B_2 \ldots B_n$ is a rule in P with $n \geq 0\}$;
  $\delta(t, a, A') = \{(t, B_1 B_2 \ldots B_n') : S \to a B_1 B_2 \ldots B_n$ is a rule in P with $n > 0\} \cup \{(u, \varepsilon) : A \to a$ is a rule in $P\}$;
  $\delta(q, v, A), \delta(q, v, A')$ are $\emptyset$ for all states q, $A \in N$ and v where not defined otherwise before; note that in a righthand side of a rule, only the last symbol can be primed.

Exercise 7.16. *The above algorithm can be made much simpler in the case that ε is not in the language. So given a grammar (N, Σ, P, S) in Greibach Normal Form for a language L with $\varepsilon \notin L$, show that there is a pushdown automaton with non-terminals N, start symbol S, terminals Σ and accepting by empty stack; determine the corresponding transition function δ in dependence of P such that in each step, the pushdown automaton reads one non-terminal.*

Example 7.17. Consider the following pushdown automaton:

- $Q = \{s\}$; $F = \{s\}$; start state s;
- $N = \{S\}$; start symbol S;
- $\Sigma = \{0, 1, 2, 3\}$;

- $\delta(s, 0, S) = \{(s, \varepsilon)\}$;
  $\delta(s, 1, S) = \{(s, S)\}$;
  $\delta(s, 2, S) = \{(s, SS)\}$;
  $\delta(s, 3, S) = \{(s, SSS)\}$;
  $\delta(s, \varepsilon, S) = \emptyset$;
- Acceptance mode is by empty stack.

The language recognised by the pushdown automaton can be described as follows: Let $digitsum(w)$ be the sum of the digits occurring in w, that is, $digitsum(00203)$ is 5. Now the automaton recognises the following language:

> $\{w : digitsum(w) < |w|$ and all proper prefixes v of w with satisfy $digitsum(v) \geq |v|\}$.

This pushdown automaton has one property: In every situation there is exactly one step the pushdown automaton can do, so it never has a non-deterministic choice. Thus the run of the pushdown automaton is deterministic.

Exercise 7.18. *Provide context-free grammars generating the language of Example 7.17 in Greibach Normal Form and in Chomsky Normal Form.*

Note that languages which are recognised by deterministic pushdown automata can be recognised in linear time, that is, time $O(n)$. For that reason, the concept of a deterministic pushdown automaton is quite interesting. It is much more flexible, if one uses the condition of acceptance by state rather than acceptance by empty stack; therefore it is defined as follows.

Definition 7.19. A deterministic pushdown automaton is given as $(Q, \Sigma, N, \delta, s, S, F)$ and has the acceptance mode by state with the additional constraint that for every $A \in N$ and every $v \in \Sigma^*$ and every $q \in Q$ there is at most one prefix $\tilde{v}$ of v for which $\delta(q, \tilde{v}, A)$ is not empty and this set contains exactly one pair $(p, \tilde{w})$. The languages recognised by a deterministic pushdown automaton are called deterministic context-free languages. Without loss of generality, $\delta(q, v, A)$ is non-empty only when $v \in \Sigma \cup \{\varepsilon\}$.

Proposition 7.20. *Deterministic context-free languages are closed under complement.*

Proof. Given a deterministic pushdown automaton $(Q, \Sigma, N, \delta, s, S, F)$ which has acceptance mode by state, one constructs the new automaton as follows:

- $Q' = Q \cup \{t, u\}$ for new state t, u; $F' = \{u\} \cup Q - F$; new start state is t;

- the terminal alphabet Σ remains the same;
- $N' = N \cup \{S'\}$ for a new start symbol S';
- The new transition function δ' is as follows, where $v \in \Sigma \cup \{\varepsilon\}$, $a \in \Sigma$, $q \in Q$, $A \in N$:
  1. $\delta'(t, \varepsilon, S') = \{(s, SS')\}$;
  2. if $\delta(q, v, A) \neq \emptyset$ then $\delta'(q, v, A) = \delta(q, v, A)$;
  3. if $\delta(q, a, A)$ and $\delta(q, \varepsilon, A)$ are both $\emptyset$ then $\delta'(q, a, A) = (u, S')$;
  4. $\delta'(q, a, S') = \{(u, S')\}$;
  5. $\delta'(u, a, S') = \{(u, S')\}$;
  6. δ' takes on all combinations not previously defined the value $\emptyset$.

The new pushdown automaton does the following:

- It starts with state t and symbol S' and pushes SS' onto the stack before simulating the old automaton by instruction of type 1;
- It then simulates the old automaton using instructions of type 2 and it accepts iff the old automaton rejects;
- When the old automaton gets stuck by a missing instruction then the new automaton pushes S' and goes to state u by instruction of type 3;
- When the old automaton gets stuck by empty stack then this is indicated by S' being the symbol to be used and the new automaton pushes S' back onto the stack and goes to state u by instruction of type 4;
- Once the automaton reaches state u and has S' on the top of the stack, it stays in this situation forever and accepts all subsequent inputs by instructions of type 5;
- The instruction set is completed by defining that δ' takes $\emptyset$ in the remaining cases in order to remain deterministic and to avoid choices in the transitions.

Note that the new automaton never gets stuck. Thus one can, by once again inverting the accepting and rejecting state, use the same construction to modify the old automaton such that it never gets stuck and still recognises the same language.

Proposition 7.21. *If L is recognised by a deterministic pushdown automaton $(Q, \Sigma, N, \delta, s, S, F)$ which never gets stuck and H is recognised by a complete deterministic finite automaton $(Q', \Sigma, \delta', s', F')$ then $L \cap H$ is recognised by the deterministic pushdown automaton*

$$(Q \times Q', \Sigma, N', \delta \times \delta', (s, s'), S, F \times F')$$

and $L \cup H$ is recognised by the deterministic pushdown automaton

$$(Q \times Q', \Sigma, N', \delta \times \delta', (s, s'), S, Q \times F' \cup F \times Q')$$

where $(\delta \times \delta')((q, q'), a, A) = \{((p, p'), w) : (p, w) \in \delta(q, a, A)$ *and* $p' = \delta'(q', a)\}$ *and* $(\delta \times \delta')((q, q'), \varepsilon, A) = \{((p, q'), w) : (p, w) \in \delta(q, \varepsilon, A)\}$.

Proof Idea. The basic idea of this proposition is that the product automaton simulates both the pushdown automaton and finite automaton in parallel and since both automata never get stuck and the finite automaton does not use any stack, the simulation of both is compatible and does never get stuck. For $L \cap H$, the product automaton accepts if both of the simulated automata accept; for $L \cup H$, the product automaton accepts if at least one of the simulated automata accepts. Besides that, both product automata do exactly the same.

Example 7.22. *There is a deterministic pushdown-automaton which accepts iff two types of symbols appear in the same quantity, say* 0 *and* 1 *and which never gets stuck:*

- $Q = \{s, t\}$*;* $\{s\}$ *is the set of accepting states;* s *is the start state;*
- Σ *contains* 0 *and* 1 *and perhaps other symbols;*
- $N = \{S, T, U, V, W\}$ *and* S *is the start symbol;*
- δ *takes non-empty output only if exactly one symbol from the input is parsed; the definition is the following:*
  $\delta(q, a, A) = \{(q, A)\}$ *for all* $a \in \Sigma - \{0, 1\}$ *and* $A \in N$*;*
  $\delta(s, 0, S) = \{(t, US)\}$*;* $\delta(s, 1, S) = \{(t, TS)\}$*;*
  $\delta(t, 1, U) = \{(s, \varepsilon)\}$*;* $\delta(t, 0, U) = \{(t, VU)\}$*;*
  $\delta(t, 1, V) = \{(t, \varepsilon)\}$*;* $\delta(t, 0, V) = \{(t, VV)\}$*;*
  $\delta(t, 0, T) = \{(s, \varepsilon)\}$*;* $\delta(t, 1, T) = \{(t, TW)\}$*;*
  $\delta(t, 0, W) = \{(t, \varepsilon)\}$*;* $\delta(t, 1, W) = \{(t, WW)\}$.

The idea is that the stack is of the form S *when the symbols are balanced and of the form* US *if currently one zero more has been processed than ones and of the form* $V^n US$ *if currently* $n+1$ *zeroes more processed than ones and of form* TS *if currently one one more has been processed than zeroes and of the form* $W^n TS$ *if currently* $n+1$ *ones more have been processed than zeroes. The state* s *is taken exactly when the stack is of the form* S *and the symbols* U, T *are there to alert the pushdown automaton that, when the current direction continues, the next symbol on the stack is* S.

Theorem 7.23. *The deterministic context-free languages are neither closed under union nor under intersection.*

Proof. If the deterministic context-free languages would be closed under union, then they would also be closed under intersection. The reason is that if L and H are deterministic context-free, then

$$L \cap H = \Sigma^* - ((\Sigma^* - L) \cup (\Sigma^* - H))$$

and so it is sufficient to show that they are not closed under intersection. By Example 7.22 there language $L_{0,1}$ of all words in $\{0,1,2\}^*$ with the same amount of 0 and 1 is deterministic context-free and so is also the language $L_{1,2}$ of all words in $\{0,1,2\}^*$ with the same amount of 1 and 2. Now assume that the intersection $L_{0,1} \cap L_{1,2}$ would be deterministic context-free. Then so is also the intersection of that language with $0^*1^*2^*$ by Proposition 7.21; however, the language

$$L_{0,1} \cap L_{1,2} \cap 0^*1^*2^* = \{0^n1^n2^n : n \in \mathbb{N}\}$$

is not context-free and therefore also not deterministic context-free. Thus the deterministic context-free languages are neither closed under union nor under intersection. ∎

Exercise 7.24. *Show that the language $L = \{0^n10^m : n \geq m\}$ is deterministic context-free. What about L^*? Give reasons for the answer, though a full proof is not requested.*

Exercise 7.25. *Assume that L is deterministic context-free and H is regular. Is it always true that $L \cdot H$ is deterministic context-free? Give reasons for the answer, though a full proof is not requested.*

Exercise 7.26. *Assume that L is deterministic context-free and H is regular. Is it always true that $H \cdot L$ is deterministic context-free? Give reasons for the answer, though a full proof is not requested.*

Exercise 7.27. *Is L^{mi} deterministic context-free whenever L is? Give reasons for the answer, though a full proof is not requested.*

Selftest 7.28. Construct a homomorphism h and a context-free set L of exponential growth such that $h(L)$ has polynomial growth and is not regular.

Selftest 7.29. Construct a homomorphism from $\{0,1,2,\ldots,9\}^*$ to $\{0,1\}^*$ are there such that

- The binary value of $h(w)$ is at most the decimal value of w for all $w \in \{0,1\}^*$;
- $h(w) \in 0^*$ iff $w \in 0^*$;

- $h(w)$ is a multiple of three as a binary number iff w is a multiple of three as a decimal number.

Note that $h(w)$ can have leading zeroes, even if w does not have them (there is no constraint on this topic).

Selftest 7.30. Consider the language L generated by the grammar $(\{S, T\}, \{0, 1, 2\}, \{S \rightarrow 0S2|1S2|02|12|T2, T \rightarrow 0T|1T\}, S)$. Provide grammars for L in Chomsky Normal Form, in Greibach Normal Form and in the normal form for linear languages.

Selftest 7.31. Carry out the Algorithm of Cocke, Kasami and Younger with the word 0122 with the grammar in Chomsky Normal Form from Selftest 7.30. Provide the table and the decision of the algorithm.

Selftest 7.32. Carry out the $O(n^2)$ Algorithm derived from the one of Cocke, Kasami and Younger with the word 0022 using the grammar in the normal form for linear grammars from Selftest 7.30. Provide the table and the decision of the algorithm.

Selftest 7.33. Provide an example of a language L which is deterministic context-free and not regular such that also $L \cdot L$ is deterministic context-free and not regular.

Selftest 7.34. Provide an example of a language L which is deterministic context-free and not regular such that $L \cdot L$ is regular.

Solution for Selftest 7.28. Let $L = \{w \in \{0,1\}^* \cdot \{2\} \cdot \{0,1\}^* : w \text{ is a palindrome}\}$ and let $h(0) = 1$, $h(1) = 1$, $h(2) = 2$. While L has exponentially many members – there are 2^n words of length $2n+1$ in L – the set $h(L) = \{1^n 2 1^n : n \in \mathbb{N}\}$ and therefore $h(L)$ has only polynomial growth, there are $n+1$ many words up to length $2n+1$ in $h(L)$. However, $h(L)$ is not regular.

Solution for Selftest 7.29. One can define h as follows: $h(0) = 00$, $h(1) = 01$, $h(2) = 10$, $h(3) = 11$, $h(4) = 01$, $h(5) = 10$, $h(6) = 11$, $h(7) = 01$, $h(8) = 10$, $h(9) = 11$. Then one has that

$$\begin{aligned} binval(h(a_n a_{n-1} \ldots a_1 a_0)) &= \sum_m 4^m binval(h(a_m)) \\ &\leq \sum_m 4^m \cdot a_m \leq decval(a_n a_{n-1} \ldots a_1 a_0) \end{aligned}$$

which gives the constraint on the decimal value. Furthermore, when taking modulo 3, 10^m and 4^m are 1 modulo 3 and a_m and $h(a_m)$ have the same value modulo three, thus they are the same. In addition, as only $h(0) = 00$, it follows that $h(w) \in 0^*$ iff $w \in 0^*$.

Solution for Selftest 7.30. The non-terminal T in the grammar is superfluous. Thus the grammar is $(\{S\}, \{0,1,2\}, \{S \to 0S2|1S2|02|12\}, S)$ and has the following normal forms:

Chomsky Normal Form: Non-terminals S, T, X, Y, Z; Terminals $0,1,2$; Rules $S \to XT|YT|XZ|YZ$, $T \to SZ$, $X \to 0$, $Y \to 1$, $Z \to 2$; start symbol S.

Greibach Normal Form: Non-terminals S, T; Terminals $0,1,2$; Rules $S \to 0ST|1ST|0T|1T$, $T \to 2$; start symbol S.

Normal Form of linear grammar: Non-terminals S, T; Terminals $0,1,2$; Rules $S \to 0T|1T$, $T \to S2|2$; start symbol S.

Solution for Selftest 7.31.

$$\begin{array}{ccccccc} & & & E_{1,4} = \{S\} & & & \\ & & E_{1,3} = \emptyset & & E_{2,4} = \{T\} & & \\ & E_{1,2} = \emptyset & & E_{2,3} = \{S\} & & E_{3,4} = \emptyset & \\ E_{1,1} = \{X\} & & E_{2,2} = \{Y\} & & E_{3,3} = \{Z\} & & E_{4,4} = \{Z\} \\ 0 & & 1 & & 2 & & 2 \end{array}$$

As $S \in E_{1,4}$, the word 0122 is generated by the grammar.

Solution for Selftest 7.32.

$$
\begin{array}{ccccccc}
 & & & E_{1,4} = \{S\} & & & \\
 & & E_{1,3} = \emptyset & & E_{2,4} = \{T\} & & \\
 & E_{1,2} = \emptyset & & E_{2,3} = \{S\} & & E_{3,4} = \emptyset & \\
E_{1,1} = \emptyset & & E_{2,2} = \emptyset & & E_{3,3} = \{T\} & & E_{4,4} = \{T\} \\
0 & & 0 & & 2 & & 2
\end{array}
$$

As $S \in E_{1,4}$, the word 0022 is generated by the grammar.

Solution for Selftest 7.33. The following example L is non-regular and deterministic context-free: $L = \{0^n 1^n : n > 0\}$. Now $L \cdot L = \{0^n 1^n 0^m 1^m : n, m > 0\}$ is also deterministic context-free.

Solution for Selftest 7.34. The following example provides a non-regular and deterministic context-free L such that $L \cdot L$ is regular: $L = \{0^n 1 0^m : n \neq m\}$. Now $L \cdot L = \{0^n 1 0^k 1 0^m : k \geq 2$ or ($k = 1$ and $n \neq 0$ and $m \neq 1$) or ($k = 1$ and $n \neq 0$ and $m \neq 1$) or ($k = 0$ and $n \neq 0$ and $m \neq 0$)$\}$, thus L is regular. Let a word $0^n 1 0^k 1 0^m$ be given. If $k \geq 2$ there are at least three ways to write k as a sum $i + j$ and at least one way satisfies that $n \neq i$ and $j \neq m$ and $0^n 1 0^k 1 0^m \in L \cdot L$; for $k = 1$ there are only two ways and it is coded into the condition on $L \cdot L$ that one of these ways has to work; for $k = 0$ it is just requiring that n, m are both different from 0 in order to achieve that the word $0^n 1 0^k 1 0^m$ is in $L \cdot L$.

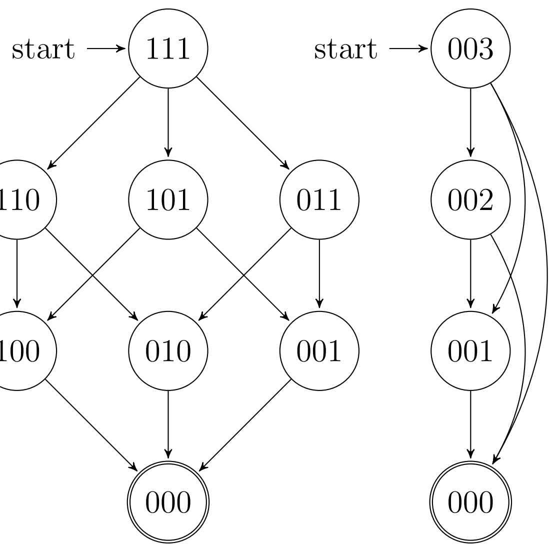
8 Games on Finite Graphs

The easiest game in a finite graph is a race to reach a member of a set of targets. Two players, Anke and Boris, move a marker alternately and that player who moves the marker first into a target-node wins the game, the game is supposed to start outside the set of targets. Without loss of generality, Anke is the player who moves first. So a game is given by a graph with a special set of vertices being the targets plus a starting-position of the marker (unless that is random). Furthermore, one might say that if a player ends up being unable to move, this player also loses the game.

Example 8.1. Consider a graph whose vertices are all labeled with 3-digit figures, so with $000, 001, \ldots, 999$, the start point is random. Now a player can move from ijk to $i'j'k'$ iff two digits are the same and the third digit is smaller than the previous one; the player who moves into 000 is the one who wins the game. The players move alternately.

Assume that 257 is the randomly chosen starting configuration. Now Anke moves $257 \to 157$. Boris replies by $157 \to 156$. Anke now moves $156 \to 116$. Boris replies $116 \to 110$. Now Anke moves $110 \to 100$ and Boris wins moving $100 \to 000$.

Assume now that 111 is the randomly chosen starting configuration. Then Anke wins: in each move, the number of 1s goes down by 1 and so Anke, Boris and then Anke can move where Anke reaches 000. For example Anke: $111 \to 110$; Boris: $110 \to 010$; Anke: $010 \to 000$. The game has a quite large graph, here just the small parts of the last play and the next one.



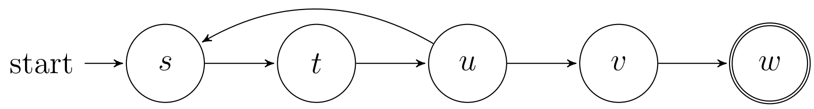
In the second play, starting at 003, Anke could win by $003 \to 000$; if she plays $003 \to 002$, Boris could win or make a bad move as well. So it depends on the move which player wins.

Definition 8.2: Winning Positions and Winning Strategies. *A winning strategy is an algorithm or table which tells Anke in each position how to move (in dependence of the prior moves which occurred in the game) such that Anke will eventually win. A node v is a winning position for Anke iff there is a winning strategy which tells Anke how to win, provided that the game starts from the node v. Similarly one defines winning strategies and positions for Boris.*

Example 8.3. In the game from Example 8.1, the node 111 is a winning position for each of the player (when it is his or her turn). The node 012 is also a winning position, as the player (whose turn it is) moves to 011; the opponent can only either move to 010 or 001 after which the player wins by moving to 000.

Exercise 8.4. *Consider the game from Example 8.1 and the following starting positions:* 123, 232, 330, 333. *Which of these starting positions are winning positions for Anke and which of these are winning positions for Boris? Explain the answer.*

Example 8.5. Consider the following game:



Each player who is in u cannot go directly to w but only to v; if the player decides, however, to go to v then the opponent would reach w and win the game. Therefore, if the player does not want to lose and is in u, the player would have to move to s. Thus the nodes s, t, u are not a winning position for either player, instead in these positions the player can preserve a draw. Such a draw might result in a play which runs forever; many board games have special rules to terminate the game as draw in the case that a situation repeats two or three times.

Several games (like the above) do not have that the starting position is a winning position for either player. Such games are called draw games.

Theorem 8.6. *There is an algorithm which determines which player has a winning strategy. The algorithm runs in time polynomial in the size of the graph.*

Proof. Let Q be the set of all nodes and T be the set of target nodes. The games starts in some node in $Q - T$.

1. Let $T_0 = T$ and $S_0 = \emptyset$ and $n = 0$.

2. Let $S_{n+1} = S_n \cup \{q \in Q - (T_n \cup S_n) :$ one can go in one step from q to a node in $T_n\}$.
3. Let $T_{n+1} = T_n \cup \{q \in Q - (T_n \cup S_{n+1}) :$ if one goes from q one step then one ends up in $S_{n+1}\}$.
4. If $S_{n+1} \neq S_n$ or $T_{n+1} \neq T_n$ then let $n = n + 1$ and goto 2.
5. Now S_n is the set of all winning positions for Anke and $T_n - T$ is the set of all winning positions for Boris and the remining nodes in $Q - (T_n \cup S_n)$ are draw positions.

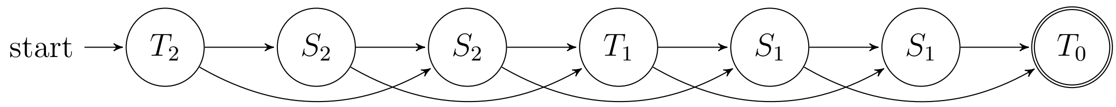
One can see that the algorithm terminates, as it can run the loop from steps 2 to 4 only as long as it adds nodes to the sets S_n or T_n, hence it runs at most $|Q|$ times through the loop.

Now one shows by induction that every set S_n consists of winning positions for Anke and T_n of winning positions for Boris. Clearly the nodes in S_1 permit Anke to win in one move. If Anke has to move from a node in $T_1 - T_0$ then she can either only move to nodes in S_1 or cannot move at all; in the first case, Boris wins the game (by symmetry), in the second case Anke loses the game as she cannot move.

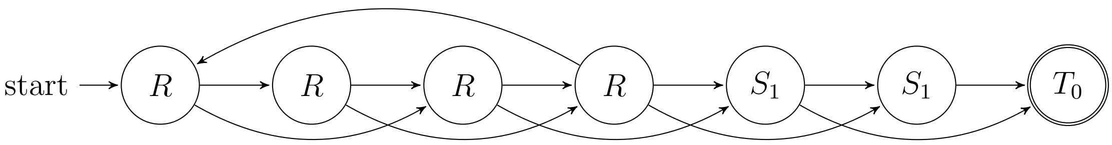
Assume now that the S_n consists of winning-positions for Anke and T_n of losing-positions for her, that is, winning-positions for Boris. Now S_{n+1} is the set of all nodes on which Anke can go into a node in T_n, that is, either Anke would win the game directly or Boris would lose it when continuing to play from that position. Hence every node in S_{n+1} is a winning-position for Anke. Furthermore, every node in T_{n+1} is a losing-position for Anke, for the nodes in T_n this is true by induction hypothesis and for the nodes in $T_{n+1} - T_n$, Anke can only move into a node in S_{n+1} from which on then Boris would win the game.

Hence, by induction, the final sets S_n are all winning positions for Anke and T_n are all winning-positions for Boris. Consider now any position in $q \in R$ with $R = Q - S_n - T_n$. Each node in R has at least one successor in R and every successor of it is either in R or in S_n. Hence the player (whether Anke or Boris) would move to a successor node in R and avoid going to S_n so that the opponent cannot win the game; as a result, the marker would circle indefinitely between the nodes in R.

The proof is illustrated by the following two examples of graphs. The nodes are labelled with the names of the sets to which they belong, therefore several nodes can have the same label, as they belong to the same set.

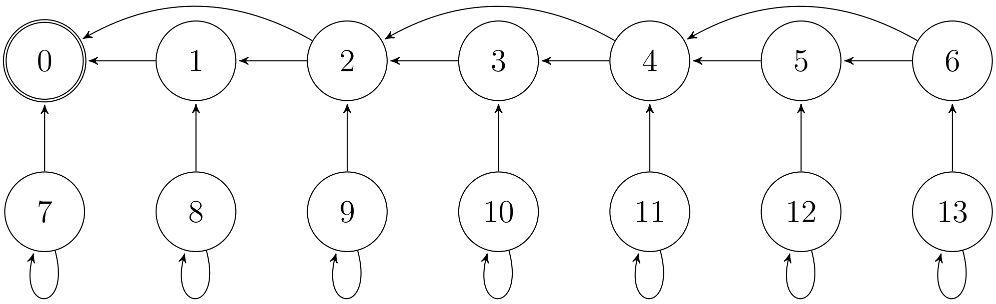


So the above game is a losing game for Anke and a winning game for Boris.



Here the players will always move inside the set R of nodes and not move to the nodes of S_1 as then the opponent wins. ▮

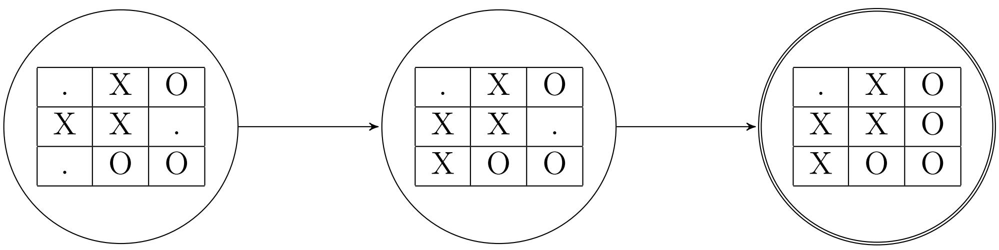
Exercise 8.7. *Consider a graph with node-set $Q = \{0, 1, 2, \ldots, 13\}$, target $T = \{0\}$ and the following edges between the nodes.*



Determine which of the nodes $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$ *are winning-positions, losing-positions and draw-positions for player Anke.*

Example 8.8: Tic Tac Toe. Tic Tac Toe is a board game with a $3 * 3$ board. Anke and Boris place alternately an X and an O on the board until either there are three times the same symbol in a row, diagonal or column or all fields are full. In the case that a player makes the three symbols in a row / diagonal / column full, this player wins. Otherwise the game is a draw.

One can represent this as a game on a graph. Each possible board configuration represents a node and one makes an edge from one node to another one if the board of the second node is obtained from the board of the first node by placing one X or O into one field, respectively; furthermore, if there are as many X as O, an X has to be placed. If there are more X than O, an O has to be placed.



There are two more conditions to be taken care off: The starting configuration is the empty board and there is no outgoing edge in the case that the target has already been reached, that is, three symbols of the same type are already in a row / diagonal / column.

It should be noted that there are two types of nodes: Those nodes which equally many X and O are the nodes where Anke moves and places an X, those nodes with more X than O are the nodes where Boris moves and places an O.

One might ask how many nodes one needs to represent the game. An upper bound is certainly $3^9 = 19683$ which is the number of all $3 * 3$ boards with a space, X or O on each cell. So the graph of the game can easily be analysed by a computer. Furthermore, the number of nodes is even smaller as there are many cases which do not occur in a play, for example a board where there are all X in the top row and all O in the bottom row or a board with 5 X and 2 O in total. Indeed, the graph of the game has been analysed well and it has been shown that the game is a draw game; good human players can also always obtain a draw.

Description 8.9: Board Games. Many games have a default starting position and are played by two players with alternating moves where the set of possible configurations is finite, that is, such games can be represented as a graph game as done for Tic Tac Toe above. Traditionally, for board games with pieces, the starting player Anke has white pieces in the board game and the second player Boris the black pieces, so they are sometimes also referred as "White" or "First Player" and "Black" or "Second player", respectively. There are now three possible outcomes: The first player always wins the game when playing optimally, the second player always win the game when playing optimally, the game always ends up in a draw when both players play optimally. For some famous games it has been computed which player can force a win or whether the game ends up in a draw, see `http://en.wikipedia.org/wiki/Solved_game` for a current list of solved games and descriptions of the corresponding games. Here just an overview with the most famous games:

- The first player wins: Connect Four, Hex (on $n * n$ board), $15 * 15$ Gomoku (no opening rules).
- The second player wins: $4 * 4$ Othello, $6 * 6$ Othello.
- Draw: Draughts (also known as Checkers), Nine Men's Morris, Tic Tac Toe.
- Unknown: Chess, Go, $19 * 19$ Gomoku (conjecture: second player wins), $8 * 8$ Othello (conjecture: draw).

Gomoku is played on both sizes, $15 * 15$ and $19 * 19$; the latter is popular due to being the size of a Go board. The Wikipedia page on Gomoku has an example with a $15 * 15$

board. Opening rules are introduced in order to balance the chances for the winner and computers were not able to solve the balanced version of the $15 * 15$ Gomoku so far.

Furthermore, $8*8$ is the usual board size for Othello. Also this game has only been solved for smaller board sizes, so one does not know how the game would behave on the traditional size. Although an algorithm is known in principle, it uses up too much resources (computation time and space) to run on current computers. Nevertheless, computers can for some still unsolved games like chess compute strategies which are good although not optimal; such computer programs can defeat any human player, even the world chess champion Garry Kasparov was defeated by a computer in a tournament of six games in 1997.

Games involving random aspects (cards, dices, ...) do not have perfect strategies. The reason is that a move which is good with high probability might turn out to be bad if some unlikely random event happens. Nevertheless, computers might be better than humans in playing these games.

Multiplayer games with more than two players usually do not have winning strategies as at three players, two of them might collaborate to avoid that the third player wins (although they should not do it). So it might be impossible for a single player to force a win.

Therefore the above analysis was for games with two players games without random aspects. If there is just a random starting point (in the graph), but no other random event, one can determine for each possible starting point which player has a winning strategy when starting from there.

Exercise 8.10. *Let $Divandinc_{n,m}$ be given by the graph with domain $\{1, 2, \ldots, n\}$, starting state $m \in \{2, \ldots, n\}$ and target state 1. Furthermore, each player can move from $k \in \{2, \ldots, n\}$ to $\ell \in \{1, 2, \ldots, n\}$ iff either $\ell = k + 1$ or $\ell = k/p$ for some prime number p. Hence the game is called "$Divandinc_{n,m}$" (Divide and increment).*

(a) Show that there is no draw play, that is, whenever the game goes through an infinite sequence of moves then some player leaves out a possibility to win.

(b) Show that if $m \leq n \leq n'$ and n is a prime number, then the player who can win $Divandinc_{n,m}$ can also win $Divandinc_{n',m}$.

(c) Find values m, n, n' with $m < n < n'$ where Anke has a winning strategy for $Divandinc_{n,m}$ and Boris for $Divandinc_{n',m}$.

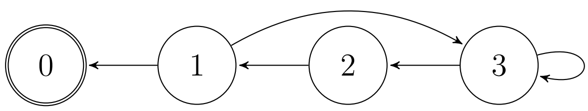
Remark 8.11. One can consider a variant of the game on finite graphs satisfying the following constraints:

- The set of nodes is partitioned into sets A, B such that every node is in exactly one of these sets and player Anke moves iff the marker is in A and player Boris moves iff the marker is in B;

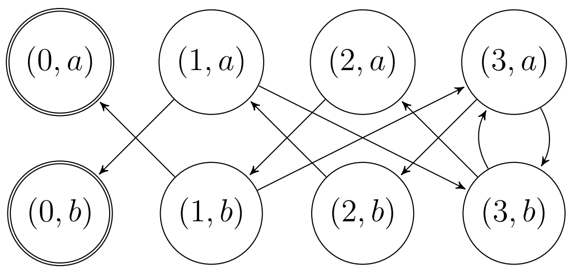
- There are three different disjoint sets T_A, T_B, T_D of target nodes and Anke wins whenever the game reaches one of the nodes in T_A and Boris wins whenever the game reaches one of the nodes in T_B and the game is draw when it ends up in T_D. Furthermore, a node is in $T_A \cup T_B \cup T_D$ iff it has no outgoing edges.

Tic Tac Toe, as described above, satisfies both constraints. A $3 * 3$ board is in A iff there are as many Xs as Os; a $3 * 3$-board is in B iff there are one X more than Os. Furthermore, T_A contains those boards from $A \cup B$ for which there is at least one row / diagonal / column with three Xs and none with three Os. T_B contains those boards from $A \cup B$ for which there is at least one row / diagonal / column with three Os and none with three Xs. T_D contains those $3 * 3$-boards where every field is either O or X but where there is no row, column or diagonal with all three symbols being the same. Boards with rows for both / diagonals / columns of three own symbols for both players cannot be reached in a play starting at the empty board without going through $T_A \cup T_B$, thus one can ignore those.

Example 8.12. Assume that a game with states Q and target set T is given. Now one can consider a new game with nodes $Q \times \{a, b\}$, there are the edges $(p, a) \to (q, b)$ and $(p, b) \to (q, a)$ in the new game whenever the edge $p \to q$ exists in the old game, $T_A = T \times \{b\}$, $T_B = T \times \{a\}$. Now $q_0 \to q_1 \to q_2 \to \ldots \to q_{2n} \to q_{2n+1}$ is a play in the old game iff $(q_0, a) \to (q_1, b) \to (q_2, a) \to \ldots \to (q_{2n}, a) \to (q_{2n+1}, b)$ is a play in the new game. Furthermore, that play is a winning play for Anke in the old game, that is, $q_{2n+1} \in T$ and no node before is in T, iff it is a winning play for Anke in the new game, that is, $(q_{2n+1}, a) \in T_A$ and no node before is in $T_A \cup T_B$. One can now show the following: A node $q \in T - Q$ is a winning position for Anke in the old game iff (q, a) is a winning position for Anke in the new game.



The above graph with $T = \{0\}$ is translated into the below one with $T_A = \{(0, b)\}$, $T_B = \{(0, a)\}$.



As $1, 3$ are the winning positions and 2 is a losing position in the old game, $(1, a), (3, a)$ are now winning positions and $(2, a)$ is a losing position for Anke in the new game. Furthermore, Boris wins from $(1, b), (3, b)$ and loses from $(2, b)$.

Exercise 8.13. *Design a game with A, B being disjoint nodes of Anke and Boris and the edges chosen such that*

- *the players move alternately;*
- *the sets T_A, T_B of the winning nodes are disjoint;*
- *every node outside $T_A \cup T_B$ has outgoing edges, that is, $T_D = \emptyset$;*
- *the so designed game is not an image of a symmetric game in the way it was done in the previous example.*

Which properties of the game can be used to enforce that?

Exercise 8.14. *The following game satisfies the second constraint from Remark 8.11 and has an infinite game graph.*

Assume that $Q = \mathbb{N}$, $x+4, x+3, x+2, x+1 \to x$ for all $x \in \mathbb{N}$ with the exception that nodes in T_A and T_B have no outgoing edges where $T_A = \{0, 6, 9\}$ and $T_B = \{5, 7, 12, 17\}$.

If the play of the game reaches a node in T_A then Anke wins and if it reaches a node in T_B then Boris wins. Note that if the game starts in nodes from T_A or T_B then it is a win for Anke or Boris in 0 moves, respectively.

Determine for both players (Anke and Boris) which are the winning positions for them. Are there any draw positions?

Alternation. Assume that an nfa is given and two players Anke and Boris. They process an input word w by alternatingly doing a move. Anke wins if the nfa is after the moves in an accepting state, Boris wins if the nfa is after the moves in a rejecting state.

This alternation game has been investigated in automata theory. However, it turned out that it is for many applications better if the two players do not move alternatingly but if there is an indication depending on state and character who moves. This gives rise to the notion of an alternating automaton.

Definition 8.15. *An alternating finite automaton (afa) is a finite automaton where for each pair q, a there are the following three possibilities:*

- *On (q, a) there is exactly one successor state q'. Then the automaton goes to q'.*
- *On (q, a) there is a disjunctive list like $q' \vee q'' \vee q'''$ of successor states. Then player Anke picks the successor among q', q'', q'''.*

- *On (q,a) there is a conjunctive list like $q' \wedge q'' \wedge q'''$ of successor states. Then player Boris picks the successor among q', q'', q'''.*

Anke wins a play of the automaton on a word w iff the automaton after all characters being processed is in an accepting state. An alternating automaton accepts a word w iff Anke has a winning strategy for w, that is, if Anke can win the game independent of whatever moves Boris makes when it is his turn to choose.

Example 8.16. Consider the alternating finite automaton with states $\{p, q, r\}$, alphabet $\{0, 1\}$ and the transition-rules as given by the following table:

state	type	0	1
p	start, rejecting	$p \wedge q \wedge r$	$q \vee r$
q	accepting	$p \wedge q \wedge r$	$p \vee r$
r	accepting	$p \wedge q \wedge r$	$p \vee q$

This alternating finite automaton accepts all words which ends with 1. To see this, consider any word w ending with 1. When the last 1 comes up, it is Anke to move. If the current state is p, she can either move to q or r, both are accepting; if the current state is q, she moves to the accepting state r; if the current state is r, she moves to the accepting state q. The empty word is rejected and a word with 0 is rejected as well. When the last digit 0 comes up, Boris moves and he can in all three cases choose the rejecting state p.

One can simulate an afa by an nfa. The idea is similar to Büchi's construction. Given an afa with a state Q of states, the states of the new nfa are the subsets of Q. When the nfa is in a state P, then the new state P' can be chosen by Anke as follows: For every $p \in P$, if the transition on (p, a) is to a conjunctive list $q_1 \wedge q_2 \wedge \ldots \wedge q_m$ then Anke has to put all the states $q_1, q_2, \ldots, q_m$ into P'; if the transition for (p, q) is a disjunctive list $q_1 \vee q_2 \vee \ldots \vee q_m$ then Anke can choose one of these states and put it into P'. If there is exactly one successor state (no choice), Anke puts this one into P'. The successor state for the full set P is then the list of states which Anke has put into P'. This defines a non-deterministic run and that is successful iff after processing the word all members of the state of the nfa are accepting states of the afa. This permits to state the below theorem.

Theorem 8.17. *If an afa with n states recognises a language L then there is an nfa with up to 2^n states which recognises L.*

For Example 8.16 from above, the initial state would be $\{p\}$. If a 0 comes up, the next state is $\{p, q, r\}$, if a 1 comes up, the next state would be either $\{q\}$ or $\{r\}$. In the case that the nfa is in $\{p, q, r\}$ and a 0 comes up, the nfa remains in $\{p, q, r\}$. If a 1 comes up, all the states are mapped to $\{q, r\}$ and the state is accepting. Furthermore, in the case that the state is $\{q\}$, the successor chosen on 1 is $\{r\}$; in the case that the state is $\{r\}$, the successor chosen on 1 is $\{q\}$, in the case that the state is $\{q, r\}$, the successor chosen on 1 is $\{q, r\}$ again. These non-deterministic choices guarantee that the nfa accepts whenever the run ends with a 1 and one could eliminate other possible choices in order to obtain a dfa recognising the language $\{0, 1\}^* \cdot 1$. Indeed, there is even a two-state dfa doing the same.

One might ask in general how much the blow-up is when translating an afa into a dfa. This blow-up is double-exponential although it is slightly smaller than 2^{2^n}. In fact, using Büchi's construction to make the nfa from above a dfa would then lead to a dfa whose states are sets of subsets of Q. These sets, however, do not need to contain two subsets $A, B \subseteq Q$ with $A \subset B$. The reason is that any future set of states derived from B contains as a subset a future set of states derived from A and when all those of B are accepting, the same is true for those in A. Thus, it is safe to drop from a state $P \in Powerset(Powerset(Q))$ all those members $B \in P$ for which there is an $A \in P$ with $A \subset B$. Furthermore, the afa is defined above such that it has to be complete, so in both exponentiations the empty set is not used. Thus the cardinality of the so remaining states is smaller than 2^{2^n}. The next example, however, shows that the blow-up is still very large.

Example 8.18. *There is an afa with $2n + 2$ states such that the corresponding dfa has at least 2^{2^n} states.*

Proof. The alternating finite automaton has the states s, q and $p_1, \ldots, p_n$ and $r_1, \ldots, r_n$. The alphabet is $\{0, 1, \ldots, n\}$ (all considered as one-symbol digits). The transitions are given by the following table, where $i \in \{1, \ldots, n\}$ refers in the first two rows to some arbitrary value and in the last two rows to the index of p_i and r_i, respectively.

state	0	i	$j \in \{1, \ldots, n\} - \{i\}$
s	$s \vee q$	s	s
q	$p_1 \wedge \ldots \wedge p_n$	q	q
p_i	p_i	r_i	p_i
r_i	r_i	p_i	r_i

Let L be the language recognised by the afa. It contains all words u of the form $x0y0z$ such that $x, y, z \in \{0, 1, \ldots, n\}^*$ and z contains each of the digits $1, \ldots, n$ an even number of times.

Now consider for each subset $R \subset \{1, \ldots, n\}$ the word v_R consisting of the digits occurring in R taken once and $w_R = v_R 0 v_R$. Let S be some non-empty set of such sets R and R' be a member of S and u be the concatenation of $00v_{R'}$ with all w_R such that $R \in S$. For $R \subseteq \{1, \ldots, n\}$, the following statements are equivalent:

- $v_R \in L_u$;
- $uv_R \in L$;
- there is a 0 in uv_R such that all non-zero digits after this 0 appear an even number of times;
- either $v_R 0 v_R$ belongs to the components from which u is built or $R = R'$;
- $R \in S$.

Furthermore, one can see that L_ε does not contain any v_R. Thus for each $S \subseteq Powerset(\{1, \ldots, n\})$ there is an u with $R \in S \Leftrightarrow v_R \in L_u$ for all $R \in S$ and so there are 2^{2^n} many different derivatives. Any dfa recognising L must have at least 2^{2^n} states.

Indeed, one can make a dfa with $2^{2^n} + 1$ states: It has a starting state s which it only leaves on 0. All other states are members of $Powerset(Powerset(\{1, \ldots, n\}))$. On a 0, the state s is mapped to $\emptyset$. On each further 0, the state P is mapped to $P \cup \{\emptyset\}$. On symbol $k > 0$, a state P is mapped to $\{A \cup \{k\} : A \in P \wedge k \notin A\} \cup \{A - \{k\} : A \in P \wedge k \in A\}$. A state P is accepting iff P is different from s and P is a set containing $\emptyset$ as an element. ∎

If one scales the n in the above construction such that it denotes the number of states, then the theorem says that given an afa with n states, it might be that the corresponding dfa has at least $2^{2^{n/2-2}}$ states. This is near to the theoretical upper bound 2^{2^n} which, as said, is not the optimal upper bound. So the real upper bound is between $2^{2^{n/2-2}}$ and 2^{2^n}.

Exercise 8.19. *Show that every afa with two states is equivalent to a dfa with up to four states. Furthermore, give an afa with two states which is not equivalent to any dfa with three or less states.*

Theorem 8.20. *If there are n dfas $(Q_i, \Sigma, \delta_i, s_i, F_i)$ with m states each recognising $L_1, \ldots, L_n$, respectively, then there is an afa recognising $L_1 \cap \ldots \cap L_n$ with $1 + mn$ states.*

Proof. One assumes that the Q_i are pairwise disjoint; if they were not, this could be achieved by renaming. Furthermore, one chooses an additional starting state $s \notin \bigcup_i Q_i$ and let $Q = \{s\} \cup \bigcup_i Q_i$.

On symbol a, let $s \to \delta_1(s_1, a) \wedge \ldots \wedge \delta_n(s_n, a)$; furthermore, for all Q_i and $q_i \in Q_i$, on a let $q_i \to \delta_i(q_i, a)$. In other words, in the first step Boris chooses which of the automata for the L_i he wants to track down and from then on the automaton tracks exactly this automaton; Boris can win iff the word on the input is not in the L_i chosen.

The state s is accepting iff $\varepsilon \in L_1 \cap \ldots \cap L_n$ and a state $q_i \in Q_i$ is accepting iff $q_i \in Q_i$. Thus, in the case that the word on the input is in all L_i, whatever Boris choses, Anke will win the game; in the case that the word on the input is not in some L_i, Boris can choose this L_i in the first step to track and from then onwards, the automaton will follow this language and eventually accept. ∎

Example 8.21. Let L_i contain the word with an even number of digit i and $\Sigma = \{0, 1, \ldots, n\}$, $n = 3$. Now $Q_i = \{s_i, t_i\}$, $F_i = \{s_i\}$ and if $i = j$ then $\delta_i(s_i, j) = t_i, \delta_i(t_i, j) = s_i$ else $\delta_i(s_i, j) = s_i, \delta_i(t_i, j) = t_i$.

Now $Q = \{s, s_1, s_2, s_3, t_1, t_2, t_3\}$, on 0, $s \to s_1 \wedge s_2 \wedge s_3$, on 1, $s \to t_1 \wedge s_2 \wedge s_3$, on 2, $s \to s_1 \wedge t_2 \wedge s_3$, on 3, $s \to s_1 \wedge s_2 \wedge t_3$. On j, $s_i \to \delta_i(s_i, j)$ and $t_i \to \delta_i(t_i, j)$. The states s, s_1, s_2, s_3 are accepting.

The automaton does the following on the word 2021: First, on 2, the automaton transits by $s \to s_1 \wedge t_2 \wedge s_3$; then, on 0, the automaton updates $s_1 \wedge t_2 \wedge s_2 \to s_1 \wedge t_2 \wedge s_3$; then, on 2, it updates $s_1 \wedge t_2 \wedge s_2 \to s_1 \wedge s_2 \wedge s_3$; lastly, on 1, it updates $s_1 \wedge s_2 \wedge s_2 \to t_1 \wedge s_2 \wedge s_3$.

Note that one can write the states of the dfa equivalent to a given afa as a formula of alternatingly “and” and “or” between the afa states; then, when transiting on a, one replaces the afa states in the leaves by the corresponding formula on the right side of the arrow; at the end, when all input symbols are processed, one replaces all accepting afa states by the logical constant “true” and all rejecting afa states by the logical constant “false” and then evaluates the formula. So the above formula evaluates to “false” as t_1 is a rejecting states and it only contains conjunctions.

Exercise 8.22. *If there are n nfas $(Q_i, \Sigma, \delta_i, s_i, F_i)$ with m states each recognising $L_1, \ldots, L_n$, respectively, show that there is an afa recognising $L_1 \cap \ldots \cap L_n$ with $1 + (m + |\Sigma|) \cdot n$ states.*

In particular, for $n = 2$ and $\Sigma = \{0, 1, 2\}$, construct explicitly nfas and the product afa where L_1 is the language of all words where the last letter has already appeared before and L_2 is the language of all words where at least one letter appears an odd number of times.

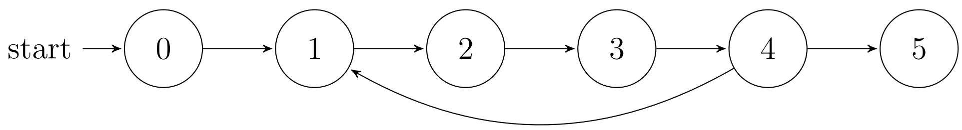
The proof can be done by adapting the one of Theorem 8.20 where one has to replace dfas by nfas. The main adjustment is that, in the first step, one goes to new, conjunctively connected states which have to memorise the character just seen, as they can not yet do the disjunction of the nfa. The next step has therefore to take care of the disjunctions of two steps of the nfa, the one of the memorised first step plus the one of the next step. From then on, all rules are now disjunctive and not deterministic as before.

9 Games Played for an Infinite Time

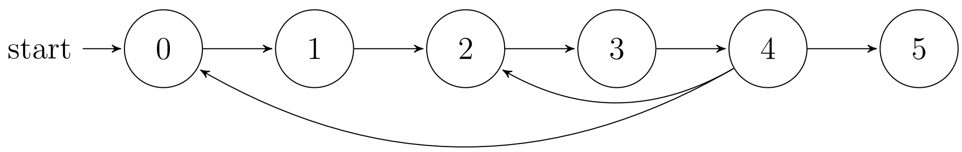
Infinite games are games where plays can go for an infinite time and nevertheless been won. A bit, the above finite games touched already this case, as for some case the draw was obtained by playing forever without going into a winning node for either player. The first type of game is a parallel situation: Anke wins iff the game runs forever.

Description 9.1: Survival Games. Mathematicians consider time often as an infinite number of steps numbered as $0, 1, 2, \ldots$; so while there is a first time 0, there is no last time and the game runs forever. Nevertheless, there is an evaluation of the overall play of the two players. Such games might be still of some interest. The most easy of these is the survival game: One player (representing a human) can influence the environment by going from one state to another, the other player (nature or weather) can modify the environment in its own way and react to the human. Both player move alternating and the Human wins if he can avoid the bad nodes (representing unacceptable environment conditions) all the time. One can represent the bad nodes by nodes without outgoing edge; then the goal of the first player (Anke) would be to be able to move as long as possible while the goal of the second player (Boris) would be that the game after some time ends up in a node without outgoing edge.

Example 9.2. Anke and Boris move alternately in the following game. Anke starts in node 0. The game has a winning strategy for Anke, as it will always be her turn to move when the game is in node 4 and Boris cannot force the game to move into the dead end 5. The reason is that Anke will always move on nodes with even numbers and Boris on nodes with odd numbers.



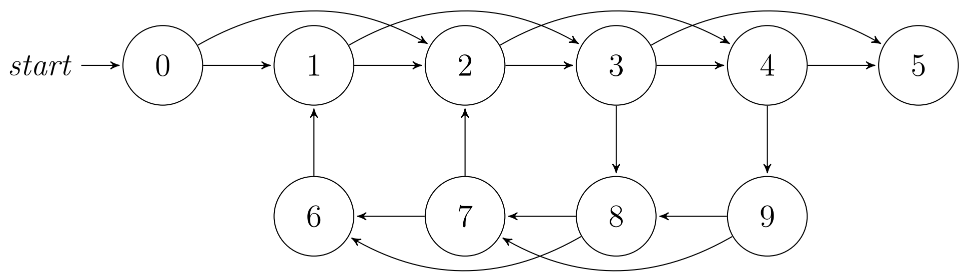
The next game is a slide modification of the above. Here now, when Anke wants is in node 4, she can only move back to 0 or 2 so that Boris gets to move on even numbers while she will end up on moving on odd numbers. So the next time Boris has the choice to move on node 4 and can terminate the game by moving into 5.



If one writes the possible plays with Boris moving into 5 whenever possible, then the plays which arise in a way consistent with this strategy are $0-1-2-3-4-5$ (where Anke gives up), $0-1-2-3-4-0-1-2-3-4-5$ (where Anke moves from 4 to 0 and then Boris moves from 4 to 5 at the next time), $0-1-2-3-4-2-3-4-5$ (where Anke moves from 4 to 2 and then Boris moves form 4 to 5 at the next time).

The winning strategy of Boris is memoryless: That is, whenever Boris makes a move, he does not have to consider the past, he has only to use the information in which node the game currently is. One can show that for a survival game, either Anke or Boris have always a memoryless winning strategy.

Quiz 9.3. *Consider the following game. Here Boris wins if the player reaches node 5 and Anke wins if the game never reaches this node.*



Which player has a winning strategy? Give a memoryless winning strategy for the player.

Theorem 9.4. *There is an algorithm which can check which player wins a survival game (when playing optimally). The algorithm runs in polynomial time.*

Proof. Let V be the set of vertices of the graph. Now one constructs a function f with domain $V \times \{\text{Anke}, \text{Boris}\}$ which tells for every node and every player whether the game will be lost for Anke within a certain amount of moves.

Make the partial function f as follows: $f(v, \text{Anke}) = 0$ or $f(v, \text{Boris}) = 0$ if the node v has no outgoing edge. Having once defined this, one extends the definition in rounds $n = 1, 2, \ldots, 2 \cdot |V|$ as follows: For each $v \in V$, if the value $f(v, \text{Anke})$ is still undefined and every outgoing edge $v \to w$ satisfies that $f(w, \text{Boris}) < n$ then let $f(v, \text{Anke}) = n$; if the value $f(v, \text{Boris})$ is still undefined and there is an outgoing edge $v \to w$ with $f(w, \text{Anke}) < n$ then let $f(v, \text{Boris}) = n$.

After $2 * |V|$ rounds, all values of f which can be defined in this way are defined, so further rounds would not add further values. Therefore, one now says that $f(v, \text{player}) = \infty$ for the remaining, not yet defined entries.

Now it is shown that the function f can be used to implement a memoryless winning strategy for that player who can win the game.

Let s be the starting node. If $f(s, \text{Anke}) = \infty$ then Anke has a winning strategy.

Each time, when its Anke's turn to move, she moves from the current node v with $f(v, \text{Anke}) = \infty$ to a node w with $f(w, \text{Boris}) = \infty$; if such a w would not exist then

$$f(v, \text{Anke}) < \max\{1 + f(w, \text{Boris}) : v \to w \text{ is an edge in the graph.}\} < \infty$$

in contradiction to the assumption on $f(v, \text{Anke})$. Now, Boris cannot move from w to any node u where $f(u, \text{Anke}) < \infty$, hence Boris moves to a node u with $f(u, \text{Anke}) = \infty$. So Anke can play in a way that the f remains on the value ∞ and will not end up in a node without outgoing edge. This strategy is obviously memoryless.

In the case that $f(s, \text{Anke}) < \infty$, Boris could play the game in a way that it takes at most $f(s, \text{Anke})$ moves. When the game is in a node with f taking the value 0, it has terminated; so consider the case that the value is larger than 0. If it is Anke's turn, she can only move from a node v to a node w with $f(w, \text{Boris}) < f(v, \text{Anke})$. If it is Boris' turn and $0 < f(v, \text{Boris}) < \infty$ then he can move to a node w with $f(w, \text{Anke}) < f(v, \text{Boris})$, so again the f value goes down. Also this strategy for Boris is obviously memoryless.

It is easy to see that the algorithm goes only through $2 * |V|$ rounds and in each round checks for $2 * |V|$ entries whether a certain condition is satisfied; this condition needs to follow all edges originating from v; therefore the overall algorithm is in $O(|V|^3)$, hence polynomial time. Note that this algorithm is not optimised for its runtime and that the cubic bound is not optimal; it is a special case of the algorithm in Theorem 8.6. ∎

Exercise 9.5. *Consider the following game $G(p, q, u, v, w)$ which is played on a graph G with u, v, w being vertices and $p \in \{Anke, Boris\}$ and $q \subseteq \{Anke, Boris\}$. Anke wins a play in this game if the game starts in node u and player p starts to move and the player moves alternately and the game goes through node v at some time and the game ends after finitely many steps in w with a player in the set q being the next to move.*

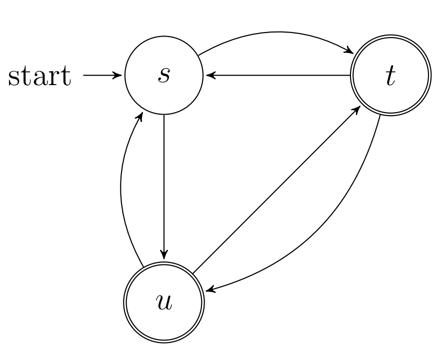
Note that if $q = \{Anke, Boris\}$ then the last condition on the last player to move is void. Furthermore, going through v includes the possibility that v is the first or last node to be visited so that the constraint on v is void in the case that $v = u$ or $v = w$. Furthermore, the graph might have more nodes than u, v, w; u, v, w are just the nodes mentioned as parameters of the game.

Give an algorithm to determine which player in this game has a winning strategy.

Description 9.6: Update Games. An update game is given by a graph with vertices V and edges E and a set W of special nodes such that Anke wins a game iff she visits during the infinite play each node in W infinitely often. Update games are so a special form of survival games, as they do not only require Anke to survive forever, but also to visit the nodes in W infinitely often; if the game gets stuck somewhere or runs forever without visiting every node in W infinitely often, then Boris wins.

Such a game might, for example, model a maintenance task where the maintenance people have to visit various positions regularly in order to check that they are in order and where various constraints — moves by own choice (player Anke) and moves imposed by other conditions beyond their control (player Boris) — influence how they navigate through this graph.

Example 9.7. Update games do not necessarily have memoryless strategies, as the following example shows (where the nodes in W are those with double boundaries).



Anke starts the game in s; whenever she moves to t or u, Boris moves the game back into s. Now, if Anke would have a memoryless strategy, she would always move to one of the two nodes in W, say to t, but then the other node in W, here u, will never be visited. So Anke has no memoryless winning strategy.

She has, however, a winning strategy using memory. Anke moves from node t to u, from node u to t and from node s to that node of u and t which has longer not yet been visited. So if Anke has to move in node s, she remembers from which node the game came to s. If the previous move (of Boris) was $t \to s$ then Anke moves $s \to u$ else Anke moves $s \to t$. This makes sure that after each visit of u, the node t is visited within 2 moves; furthermore, after each visit of t, the node u is visited within 2 moves.

Theorem 9.8. *There is an algorithm which determines which player has a winning strategy for an update game.*

Proof. Let s be the starting node and $w_1, w_2, \ldots, w_n$ be the nodes in W.

Now one first decides the following games $G(u, v, w, p, q)$ where $u, v, w \in V$ and $p \in \{\text{Anke}, \text{Boris}\}$ and q is a non-empty subsets of $\{\text{Anke}, \text{Boris}\}$. Now Anke wins the game $G(u, v, w, p, q)$, if it starts with some player p moving from u and it runs only finitely many steps visiting the node v in between and then ends up in w with a player in q being the next one to move. There are algorithms to decide this games, similar to those given in Theorem 8.6 and Theorem 9.4; Exercise 9.5 asks to design such an algorithm.

Now Anke has a winning strategy for the game iff one of the following conditions are satisfied:

- In Case 1 there is a node v and a player $p \in \{\text{Anke}, \text{Boris}\}$ such that Anke can win the game $G(\text{Anke}, \{p\}, s, v, v)$ and for each $w \in W$ Anke can win the game $G(p, \{p\}, v, w, v)$.
- In Case 2 there is a node v such that for every $w \in W$ and every player $p \in \{\text{Anke}, \text{Boris}\}$, Anke can win the games $G(\text{Anke}, \{\text{Anke}, \text{Boris}\}, s, v, v)$ and $G(p, \{\text{Anke}, \text{Boris}\}, v, w, v)$.

By assumption, this can be checked algorithmically. The algorithm is in polynomial time.

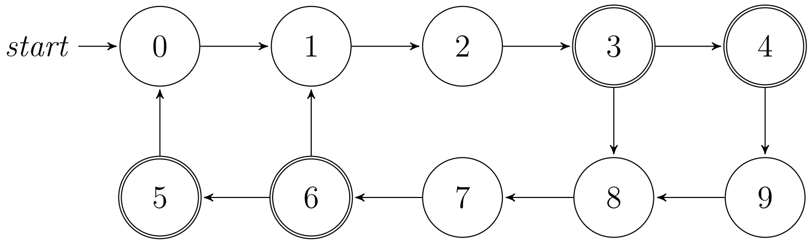
First it is verified that Anke has a winning strategy in the first case. She then can force from s that the game comes to node v and that it is player p to move. Furthermore, she can now alternatively for $w = w_1, w_2, \ldots, w_n$ force that the game visits this node and eventually returns to v with player p being the one to move. So she can force an infinite play which visits each of the nodes in W infinitely often.

Second it is verified that Anke has a winning strategy in Case 2. Anke can force the game into v without having a control which player will move onwards from v. Then, for each of the two cases, she can force that the game visits any given node $w \in W$ and returns to v, hence she can force that the game visits each of the nodes in W infinitely often.

Third assume that Case 1 and Case 2 both fail and that this is due because there is a player p and a node $u \in W$ such that Anke does not win $G(\text{Anke}, \{\text{Anke}, \text{Boris}\} - \{p\}, s, u, u)$ and Anke does not win $G(p, \{\text{Anke}, \text{Boris}\}, u, v, u)$ for some $v \in W$. Hence Boris can first enforce that either Anke visits u only finitely often or is at some point in u with the player to move being p; from then onwards Boris can enforce that the game either never reaches v or never returns to u after visiting v. Hence Boris has a winning strategy in this third case.

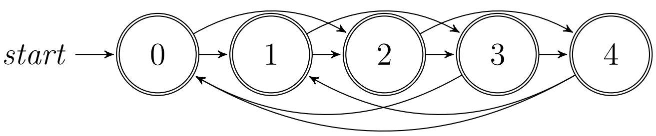
Fourth, assume that Cases 1 and 2 fail and that there are nodes $u, v, v' \in W$ such that Anke loses the games $G(\text{Anke}, \{\text{Anke}\}, u, v, u)$ and $G(\text{Boris}, \{\text{Boris}\}, u, v', u)$; hence Anke cannot enforce that a game visiting all nodes in W infinitely often has always the same player being on move when visiting u. As Case 2 fails, there is a node $w \in W$ and $p \in \{\text{Anke}, \text{Boris}\}$ such that Boris has a winning strategy for the game $G(p, \{\text{Anke}, \text{Boris}\}, u, w, u)$. It follows that once the player p is on the move in node u, Boris can enforce that either w or u is not visited again. Hence Boris can enforce that at least one of the nodes u, v, v', w is not visited infinitely often and so Boris has a winning strategy in this fourth case. This case distinction completes the proof. ∎

Quiz 9.9. *Which player has a winning strategy for the following update game?*



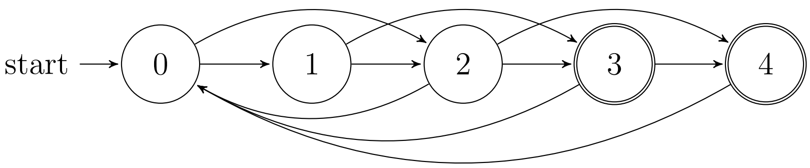
How much memory needs the strategy?

Exercise 9.10. *Let $n > 4$, n be odd, $V = \{m : 0 \leq m < n\}$ and $E = \{(m, m+1) : m < n-1\} \cup \{(m, m+2) : m < n-2\} \cup \{(n-2, 0), (n-1, 0), (n-1, 1)\}$ and $s = 0$. Show that Anke has a winning strategy for the update game (V, E, s, V) but she does not have a memoryless one.*



Here the game for $n = 5$.

Description 9.11. A Büchi game (V, E, s, W) is played on a finite graph with nodes V and edges E, starting node s and a special set $W \subseteq V$ (like an update game). Anke wins a play in the game iff, when starting in s, the game makes infinitely many moves and during these moves visits one or more nodes in W infinitely often.



Anke has a winning strategy for this game. The game is visiting node 0 infinitely often as all backward arrows end up in this node.

In the case that it is Anke's turn, she moves from 0 to 1. Then Boris can either move to 3 and visit one node in W or to 2 so that Anke in turn can move to 3 or 4 and hence visiting one of the accepting nodes.

In the case that it is Boris' move, he moves to 1 or 2 and Anke can go to the accepting node 3 from either of these nodes.

Theorem 9.12. *There is a polynomial time algorithm which decides which player can win a given Büchi game.*

Proof. Given a Büchi game (V, E, s, W), the idea is to make a function $f : V \times \{\text{Anke}, \text{Boris}\} \to \{0, 1, \ldots, 30 \cdot |V|^2\}$ which guides the winning strategies of Anke and Boris (which are memoryless).

Initialise $f(v, p) = 0$ for all nodes v and players p. The algorithm to compute f is to do the below updates as long as one of the if-conditions applies; if several apply, the algorithm does the statement of the first if-condition which applies:

- If there are nodes $v \in V - W$ and $w \in V$ with $(v, w) \in E$ and $f(v, \text{Anke}) < f(w, \text{Boris}) - 1$ then update $f(v, \text{Anke}) = f(v, \text{Anke}) + 1$;
- If there is $v \in V - W$ with an outgoing edge and all $w \in V$ with $(v, w) \in E$ satisfy $f(v, \text{Boris}) < f(w, \text{Anke}) - 1$ then update $f(v, \text{Boris}) = f(v, \text{Boris}) + 1$;
- If there are $v \in W$ with $f(v, \text{Anke}) \leq 30 \cdot |V|^2 - 3 \cdot |V|$ and $w \in V$ with $(v, w) \in E$ and $f(v, \text{Anke}) \leq f(w, \text{Boris}) + 6 \cdot |V|$ then update $f(v, \text{Anke}) = f(v, \text{Anke}) + 3 \cdot |V|$;
- If there is $v \in W$ with outgoing edge and $f(v, p) \leq 30 \cdot |V|^2 - 3 \cdot |V|$ and all $w \in V$ with $(v, w) \in E$ satisfy $f(v, \text{Boris}) \leq f(w, \text{Anke}) + 6 \cdot |V|$ then update $f(v, \text{Boris}) = f(v, \text{Boris}) + 3 \cdot |V|$.

Note that there are at most $60 \cdot |V|^3$ updates as each update increases one value of f by one and the domain has cardinality $2 \cdot |V|$ and the values in the range can be increased at most $30 \cdot |V|^2$ times. Hence the whole algorithm is polynomial in the size of $|V|$.

Now the strategy of Anke is to move from v to that node w with $(v, w) \in E$ for which $f(w, \text{Boris})$ is maximal; the strategy of Boris is to move from v to that node w with $(v, w) \in E$ for which $f(v, \text{Anke})$ is minimal.

As there are only $2 \cdot |V|$ many values in the range of f but the range can be spread out between 0 and $30 \cdot |V|^2$, there must, by the pigeon hole principle, be a natural number $m \leq 2 \cdot |V|$ such that there are no nodes v and players p with $10 \cdot |V| \cdot m \leq f(v, p) < 10 \cdot |V| \cdot (m + 1)$. Let m be the least such number. Now one shows the following claim.

Claim. If $f(v, p) \geq 10 \cdot m \cdot |V|$ then Anke can win the game else Boris can win the game.

To prove the claim, first observe that for all $w \in V - W$ with $f(w, \text{Anke}) > 0$ there is a node u with $(w, u) \in E$ and $f(u, \text{Boris}) = f(w, \text{Anke}) + 1$. The reason is that when $f(w, \text{Anke})$ was updated the last time, there was a successor $u \in V$ such that $f(u, \text{Boris}) > f(w, \text{Anke}) - 1$. Now this successor causes $f(w, \text{Anke})$ to be updated until $f(w, \text{Anke}) \geq f(u, \text{Boris}) - 1$. Similarly, for all $w \in V - W$ with $f(w, \text{Boris}) > 0$ one has that $f(w, \text{Boris}) = \min\{f(u, \text{Anke}) : (w, u) \in E\} - 1$; again, this follows from the update rules. In particular as long as the game is in $V - W$, either the values of f remain constant at 0 or they go up. As there are only finitely many values, it means that the game eventually visits a node in W whenever the f-values of the current situation in the game is positive.

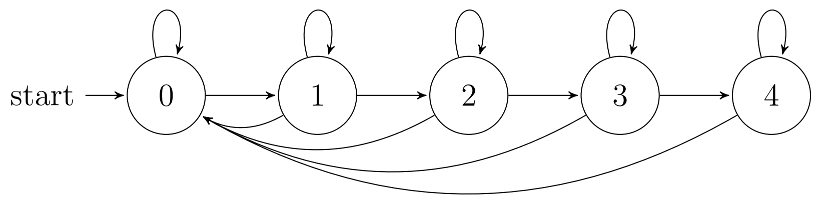
Now consider the case of any $w \in W$ and a player p moves from w to u following its strategy. If $p = $ Anke then $f(u, \text{Boris}) \geq f(w, \text{Anke}) - 6 \cdot |V|$; if $p = $ Boris then $f(u, \text{Anke}) \geq f(w, \text{Boris}) - 6 \cdot |V|$, as otherwise $f(w, p)$ would not have reached the final value in the update algorithm. Furthermore, unless $f(w, p) > 30 \cdot |V|^2 - 3 \cdot |V|$, one also can conclude that $f(w, p) \leq f(u, q) + 3 \cdot |V|$ for $q \neq p$, as otherwise a further update on $f(w, p)$ would have been done.

Thus, if $f(w, p) \geq 10 \cdot m \cdot |V|$ and the move from w to u is done in a play where Anke follows her winning strategy then actually $f(w, p) \geq 10 \cdot m \cdot |V| + 10 \cdot |V|$ and $f(u, q) \geq 10 \cdot m \cdot |V| + 10 \cdot |V|$ as well. Thus the game will go forever and visit nodes in W infinitely often.

However, consider now the case that the starting point satisfies $f(v, p) < 10 \cdot m \cdot |V|$. Then in particular $f(v, p) < 30 \cdot |V|^2 - 3 \cdot |V|$. For all (w, q) with $f(w, q) < 10 \cdot m \cdot |V|$ it holds that $f(w, q)$ cannot be increased as the conditions of the update-rules for f do no longer apply. This means that when $f(w, \text{Anke}) < 10 \cdot m \cdot |V|$ then all successor configurations satisfy the same condition; if $f(w, \text{Boris}) < 10 \cdot m \cdot |V|$ then some successor configuration satisfies the same condition. Furthermore, when $w \in W$ and $f(w, \text{Anke}) < 10 \cdot m \cdot |V|$ then all successor configurations (u, Boris) satisfy $f(u, \text{Boris}) < f(w, \text{Anke}) - 6 \cdot |V|$, if $f(w, \text{Boris}) < 10 \cdot m \cdot |V|$ and at least one successor configuration exists then there is a successor-node u with $f(u, \text{Anke}) < f(w, \text{Boris}) - 6 \cdot |V|$. Thus the play reduces each time by at least $3 \cdot |V|$ whenever it goes through a node in W. Furthermore, there is a constant c such that there is no w, p taking a value $f(w, p)$ having the remainder c modulo $3 \cdot |V|$; thus, the game cannot go up from a value of the form $3k \cdot |V|$ to $3(k + 1) \cdot |V|$ without in between visiting a node in W. Therefore the game cannot visit a node in W infinitely often when starting at (v, p) with $f(v, p) \leq 10 \cdot m \cdot |V|$.

This two case distinctions complete the proof that the function f defines a memoryless winning strategy for one of the players in the given Büchi game. ▮

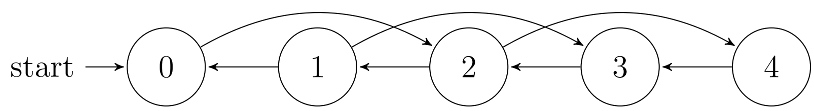
Description 9.13: Parity Games. A parity game is given by a graph (V, E) and a function val which attaches to each node $v \in V$ a value and a starting node $s \in V$. The players Anke and Boris move alternately in the graph with Anke moving first. Anke wins a play through nodes $v_0, v_1, \ldots$ in the game iff the limit superior of the sequence $val(v_0), val(v_1), \ldots$ is an even number.



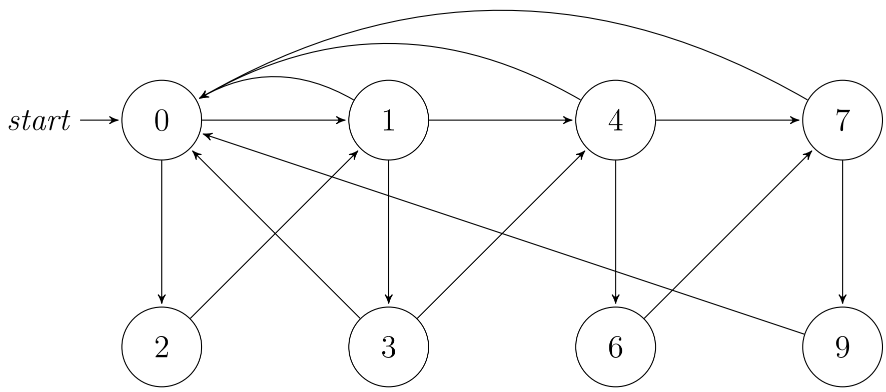
In this game, the nodes are labeled with their value, which is unique (what does not need to be). Anke has now the following memoryless winning strategy for this game: $0 \to 0$, $1 \to 2$, $2 \to 0$, $3 \to 4$, $4 \to 0$. Whenever the game leaves node 0 and Boris moves to node 1, then Anke will move to node 2. In the case that Boris moves the game into node 3, Anke will move to node 4. Hence whenever the game is in a node with odd value (what only happens after Boris moved it there), the game will in the next step go into a node with a higher even value. So the largest infinitely often visited node is even and hence the limit superior of this numbers is an even number. Hence Anke has a winning strategy for this parity game given here.

One can show that in general, whenever a player has a winning strategy for a parity game, then this winning strategy can be chosen to be memoryless; furthermore, there is always a player which has a winning strategy.

Quiz 9.14. *Which player wins this parity game? Give a winning strategy.*



Exercise 9.15. *Consider the following parity game:*



Which player has a winning strategy for this parity game? Give the winning strategy as a table (it is memoryless).

Description 9.16: Infinite games in general. The following general concept covers all the examples of games on finite graphs (V, E) with starting node s seen so far: The players Anke and Boris move alternately with Anke starting in s along the edges of the graph (which can go from a node to itself) and the winning condition consists of a function F from subsets of V to $\{\text{Anke}, \text{Boris}\}$ such that the winner of a play is $F(U)$ where U is the set of nodes visited infinitely often during the play. Here $U = \emptyset$ stands for the case that the game gets stuck after finitely many moves and no node is visited infinitely often. Here an overview how the winning conditions of the above games are translated into this general framework.

In the case of a survival game, $F(\emptyset) = \text{Boris}$ and $F(U) = \text{Anke}$ for all non-empty subsets $U \subseteq V$.

In the case of an update game with parameter W, if $W \subseteq U$ then $F(U) = \text{Anke}$ else $F(U) = \text{Boris}$.

In the case of a Büchi game with parameter W, if $W \cap U \neq \emptyset$ then $F(U) = \text{Anke}$ else $F(U) = \text{Boris}$.

In the case of a parity game, $F(\emptyset) = \text{Boris}$. For each non-empty set W, if $\max\{val(w) : w \in U\}$ is an even number (where the function val assigns to each node in V a natural number, see above) then $F(U) = \text{Anke}$ else $F(U) = \text{Boris}$.

There are games which can be captured by this framework and which are not of any of the types given above. For example, a game with $V = \{s, t, u\}$ where the players can move from each node to each node such that if $|U| = 2$ then $F(U) = \text{Anke}$ else $F(U) = \text{Boris}$.

Exercise 9.17. *Determine the function F for the following game: $V = \{0, 1, 2, 3, 4, 5\}$ the edges go from each node v to the nodes $(v + 1)$ mod 6 and $(v + 2)$ mod 6. The game starts in 0 and the players move alternately. Anke wins the game iff for each infinitely often visited node v, also the node $(v + 3)$ mod 6 is infinitely often visited.*

Define the function F on all possible values of U which can occur as an outcome of the game. List those values of U which cannot occur, that is, for which a value $F(U)$ does not need to be assigned. For example, as the game graph has for each node two outgoing edges, so it cannot get stuck somewhere and therefore $U = \emptyset$ is irrelevant.

Which player has a winning strategy for this game? Can this winning strategy be made memoryless?

Exercise 9.18. *Let a game (V, E, s, W) given by set V of nodes, possible moves E, starting node s and a set of nodes W to be avoided eventually. Let U be the infinitely often visited nodes of some play.*

Say that if $U \cap W = \emptyset$ and $U \neq \emptyset$ then Anke wins the game else Boris wins the *game.*

Determine an easy way mapping from (V, E, s, W) *to* (V', E', s', F') *and players* p *to* p' *such that player* p *wins the avoidance game* (V, E, s, W) *iff* p' *wins the game* (V', E', s', F') *(see Description 9.16) where the type of* (V', E', s', F') *is one of survival game, update game, Büchi game or parity game. Say which type of game it is and how the mapping is done and give reasons why the connection holds.*

Exercise 9.19. *Describe an algorithm which transforms a parity game* (V, E, s, val) *into a new parity game* (V', E', s', val') *such that this game never gets stuck and Anke wins* (V, E, s, val) *iff Boris wins* (V', E', s', val')*; without loss of generality it can be assumed that* V *is a finite subset of* $\mathbb{N}$ *and* $val(v) = v$*. Do the mapping such that* V' *has at most two nodes more than* V*.*

10 Automata on Infinite Sequences

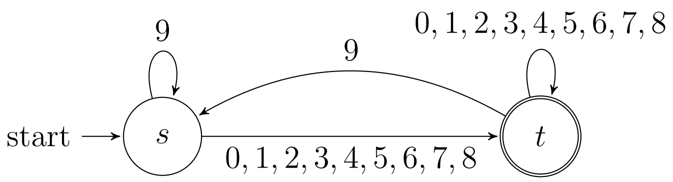
An infinite sequence $b_0b_1b_2 \ldots \in \Sigma^\omega$ is a function from $\mathbb{N}$ to Σ; such sequences are called ω-words. One can for example represent all the real numbers between 0 and 1 by ω-words with $b_0b_1b_2 \ldots \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 8\}^\omega$ representing the sum $\sum_{k \in \mathbb{N}} 10^k \cdot b_k$ where only the finite decimal fractions are not uniquely represented and have two representatives; for example $\frac{256}{1000}$ is represented by $25600000\ldots$ and $255999999\ldots$ while $\frac{1}{3}$ has the unique representative $333\ldots$ in this set. Representing real numbers is indeed one of the motivations in the study of ω-words and one can make more advanced systems which represent all numbers in $\mathbb{R}$. Richard Büchi and Lawrence Landweber [8, 9] investigated methods to deal with such representations in a way similar to what can be done with sets of finite words.

Description 10.1: Büchi Automata [8, 9]**.** A Büchi automaton is a non-deterministic automaton $(Q, \Sigma, \delta, s, F)$ where Q is the set of states, Σ the finite alphabet used, δ a set of possible transitions $(p, a, q) \in Q \times \Sigma \times Q$ such that the automaton can on symbol a go from p to q, s the starting state and F a set of states. Given an infinite word $b_0b_1b_2 \ldots \in \Sigma^\omega$, a run on this sequence is a sequence $q_0q_1q_2 \ldots \in Q^\omega$ of states such that $q_0 = s$ and $(q_k, b_k, q_{k+1}) \in \delta$ for all $k \in \mathbb{N}$. Let

$$U = \{p \in Q : \exists^\infty k \, [q_k = p]\}$$

be the set of infinitely often visited states on this run. The run is accepting iff $U \cap F \neq \emptyset$. The Büchi automaton accepts an ω-word iff it has an accepting run on this ω-word, otherwise it rejects the ω-word.

A Büchi automaton is called deterministic iff for every $p \in Q$ and $a \in \Sigma$ there is at most one $q \in Q$ with $(p, a, q) \in \delta$; in this case one also writes $\delta(p, a) = q$. The following deterministic Büchi automaton accepts all decimal sequences which do not have an infinite period of 9; that is, each real number r with $0 \leq r < 1$ has a unique representation in the set of sequences accepted by this automaton.

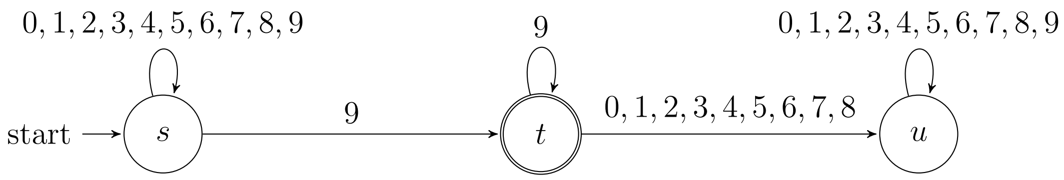


This automaton goes infinitely often through the accepting state t iff there is infinitely often one of the digits $0, 1, 2, 3, 4, 5, 6, 7, 8$ and therefore the word is not of the form $w9^\omega$.

While non-deterministic and deterministic finite automata have the same power, this is not true for Büchi automata.

Example 10.2. Let L be the language of all ω-words which from some point onwards have only 9s, so $L = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \cdot 9^\omega$. Then there is a non-deterministic but no deterministic Büchi automaton accepting L. Furthermore, the languages recognised by deterministic Büchi automata are not closed under complement.

First, one shows that a non-deterministic Büchi automaton recognises this language. So an accepting run would at some time guess that from this point onwards only 9s are coming up and then stay in this state forever, unless some digit different from 9 comes up.



Second, it has to be shown that no deterministic Büchi automaton recognises this language. So assume by way of contradiction that $(Q, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, s, F)$ would be such an automaton. Now one searches inductively for strings of the form $\sigma_0, \sigma_1, \ldots \in 09^*$ such that $\delta(s, \sigma_0\sigma_1 \ldots \sigma_n) \in F$ for all n. If there is an n such that σ_n cannot be found then the sequence $\sigma_0\sigma_1 \ldots \sigma_{n-1}09^\omega$ is not accepted by the Büchi automaton although it is in L, as states in F are visited only finitely often in the run. If all σ_n are found, then the infinite sequence $\sigma_0\sigma_1 \ldots$ has the property that it contains infinitely many symbols different from 9 and that it is not in L; however, the automaton visits infinitely often a state from F. This contradiction shows that L cannot be recognised by a deterministic Büchi automaton. ▮

Exercise 10.3. *Show the following: if L and H are languages of ω-words recognised by deterministic Büchi automata then also $L \cup H$ and $L \cap H$ are recognised by deterministic Büchi automata. Show the same result for non-deterministic Büchi automata.*

Exercise 10.4. *Make a Büchi automaton which recognises the language of all ω-words in which exactly two digits from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ occur infinitely often. Describe how the automaton is build; it is not necessary to make a complete diagram of its states.*

Büchi [8] found the following characterisation of the languages of ω-words recognised by non-deterministic Büchi automata.

Theorem 10.5: Büchi's Characterisation [8]. *The following are equivalent for a language L of ω-words:*
(a) *L is recognised by a non-deterministic Büchi automaton;*
(b) *$L = \bigcup_{m \in \{1,\ldots,n\}} A_m B_m^\omega$ for some n and $2n$ regular languages $A_1, B_1, \ldots, A_n, B_n$.*

Proof, (a)⇒(b). Assume that a non-deterministic Büchi automaton $(Q, \Sigma, \delta, s, F)$ recognises L. Let n be the cardinality of F and $p_1, p_2, \ldots, p_n$ be the states in F. Now let A_m be the language of strings recognised by $(Q, \Sigma, \delta, s, \{p_m\})$ when viewed as an nfa and B_m be the set of non-empty words in the language recognised by $(Q, \Sigma, \delta, p_m, \{p_m\})$ when viewed as an nfa. It is clear that all these languages are regular. Furthermore, for each ω-word in $A_m \cdot B_m^\omega$ there is a run of the original Büchi automaton which goes infinitely often through the state p_m. Furthermore, if a ω-word has a run which goes infinitely often through states in the set F then there is at least one state p_m which is infinitely often visited by this run; one can now partition this ω-word in parts $\sigma_0, \sigma_1, \sigma_2, \ldots$ such that the run is after processing $\sigma_0 \sigma_1 \ldots \sigma_k$ in p_m for visit number k (with the first visit having number 0). Note that all σ_k are non-empty and that $\sigma_0 \in A_m$ and each σ_k with $k > 0$ is in B_m. Hence the ω-word is in $A_m B_m^\omega$.

(b)⇒(a). Assume that $L = A_1 \cdot B_1^\omega \cup A_2 \cdot B_2^\omega \cup \ldots \cup A_m \cdot B_n^\omega$. Assume that each language A_m is recognised by the nfa $(N_{2m-1}, \Sigma, \delta_{2m-1}, s_{2m-1}, F_{2m-1})$ and each language B_m is recognised by the nfa $(N_{2m}, \Sigma, \delta_{2m}, s_{2m}, F_{2m})$.

Now let $N_0 = \{s_0\} \cup N_1 \cup N_2 \cup \ldots \cup N_{2n}$ where all these sets are considered to be disjoint (by renaming the non-terminals, if necessary). The start symbol of the new automaton is s_0. Furthermore, let δ_0 be $\delta_1 \cup \delta_2 \cup \ldots \cup \delta_{2n}$ plus the following transitions for each $a \in \Sigma$: first, (s_0, a, q) if there is an m such that $(s_{2m-1}, a, q) \in \delta_{2m-1}$; second, (s_0, a, q) if there is an m such that $\varepsilon \in A_m$ and $(s_{2m}, a, q) \in \delta_{2m}$; third, (s_0, a, s_{2m}) if $a \in A_m$; fourth, (q, a, s_{2m}), if there are m and $p \in F_{2m-1}$ with $q \in N_{2m-1}$ and $(q, a, p) \in \delta_{2m-1}$; fifth, (q, a, s_{2m}), if there are m and $p \in F_{2m}$ with $q \in N_{2m}$ and $(q, a, p) \in \delta_{2m}$. These added rules connect the transitions from the starting state into A_m (first case) or directly into B_m (when $\varepsilon \in A_m$ for the second case or $a \in A_m$ for the third case) and the transition from A_m into B_m on the reach of an accepting state of A_m (fourth case) and the return to the starting state of B_m when an accepting state can be reached (fifth case). In the fourth or fifth case, the automaton could go on in A_m or B_m instead of going to s_{2m}, this choice is left non-deterministic on purpose; indeed, one cannot in all cases make this Büchi automaton deterministic.

Now $\{s_2, s_4, s_6, \ldots, s_{2n}\}$ is the set F_0 of the final states of the Büchi automaton $(N_0, \Sigma, \delta_0, s_0, F_0)$. That is, this Büchi automaton accepts an ω-word iff there is a run of the automaton which goes infinitely often through a node of the form s_{2m}; by the way how δ_0 is defined, this is equivalent to saying that the given ω-word is in $A_m \cdot B_m^\omega$. The further verification of the construction is left to the reader. ∎

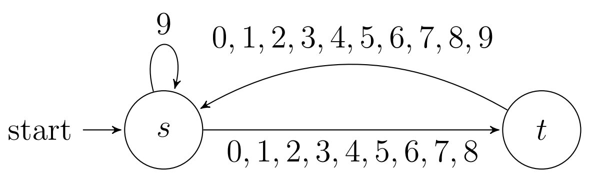
Exercise 10.6. *Make a deterministic Büchi automaton which accepts an ω-word iff it contains every digit infinitely often; for simplicity use only the digits $\{0, 1, 2\}$. Can the complement be accepted by a deterministic Büchi automaton?*

Muller introduced a notion of automata which overcomes the shortage of deterministic Büchi automata and can nevertheless be made deterministic.

Description 10.7: Muller automaton. A Muller automaton $(Q, \Sigma, \delta, s, G)$ consists of a set of states Q, an alphabet Σ, a transition relation δ, a starting state s and a set G of subsets of Q. A run of the Muller automaton on an ω-word $b_0 b_1 b_2 \ldots \in \Sigma^\omega$ is a sequence $q_0 q_1 q_2 \ldots$ with $q_0 = s$ and $(q_k, a, q_{k+1}) \in \delta$ for all k. A run of the Muller automaton is accepting iff the set U of infinitely often visited states satisfies $U \in G$. The Muller automaton accepts the ω-word $b_0 b_1 b_2 \ldots$ iff it has an accepting run on it.

A Muller automaton is deterministic iff the relation δ is a function, that is, for each $p \in Q$ and $a \in \Sigma$ there is at most one $q \in Q$ with $(p, a, q) \in \delta$.

While the language of all ω-words of the form $w9^\omega$ is not recognised by a deterministic Büchi automaton, it is recognised by the following deterministic Muller automaton: $(\{s, t\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, s, \{\{s\}\})$ where $\delta(s, a) = t$ for $t < 9$, $\delta(s, 9) = s$ and $\delta(t, a) = s$. The following diagram illustrates the Muller automaton:



$G = \{\{s\}\}$

If the ω-word consists from some time onwards only of 9s then the automaton will after that time either go to s or be in s and then remain in s forever so that $U = \{s\}$ and $U \in G$; hence the run of the automaton on this ω-word is accepting.

If the ω-word consists of infinitely many symbols different from 9 then the automaton will leave on its run infinitely often the state s for the state t and $U = \{s, t\}$ what is not in G. Hence the run of the automaton on this ω-word is rejecting.

As the automaton is deterministic, the automaton accepts a word iff it contains from some point onwards only 9s.

Exercise 10.8. *Make a deterministic Muller automaton which accepts all those ω-words in which at least one of the symbols from the alphabet $\{0, 1, 2\}$ occurs only finitely often.*

McNaughton [52] established the equivalence of these three conditions. The direct translation from (a) to (b) was also optimised by Safra [65, 66].

Theorem 10.9: McNaughton's Characterisation [52]. *The following are equivalent for a language L of ω-words:*
(a) *L is recognised by a non-deterministic Büchi automaton;*
(b) *L is recognised by a deterministic Muller automaton;*
(c) *L is recognised by a non-deterministic Muller automaton.*

Proof, (a)⇒(b). Let a non-deterministic Büchi automaton $(Q, \Sigma, \delta, s, F)$ be given. Without loss of generality, the given Büchi automaton has on every ω-word some infinite run; if not, one could add one state q such that one can go from every state into q, $q \notin F$ and one can go from q only into q itself (irrespective of the input).

The idea is that the corresponding deterministic Muller automaton keeps track of which non-terminals can be reached and how the history of visiting accepting states before the current state is. So for each initial part of a non-deterministic run $q_0 q_1 q_2 \ldots q_n$ of the Büchi automaton while processing input $b_1 b_2 \ldots b_n$, the Muller automaton machine would ideally archive the current state q_n and a string $\tau \in \{0,1\}^{n+1}$ representing the history where $\tau(m) = 0$ for $q_m \in F$ and $\tau(m) = 1$ for $q_m \notin F$. So the overall goal would be to have as many 0 as possible in the string τ and to have them as early as possible (what might be even more important than the actual number).

Hence, if there are two different runs having the traces τ and η, both ending up in the same state q_n, then the learner would only archive $\min_{lex}\{\tau, \eta\}$. For that reason, after processing $b_1 b_2 \ldots b_n$, the Muller automaton would represent the current state by a set of pairs (p, τ) with p being a state reachable on input $b_1 b_2 \ldots b_n$ and τ being the lexicographically least string representing the history of some run on this input. Note that if one can go with histories τ, τ' from s to p on $b_1 b_2 \ldots b_n$ and with histories η, η' from p to q on $b_{n+1} b_{n_2} \ldots b_m$ then one can also go with each of the histories $\tau\eta$, $\tau\eta'$, $\tau'\eta$, $\tau'\eta'$ from s to q on $b_1 b_2 \ldots b_m$. Furthermore, the lexicographic minimum of these four strings is $\min_{lex}\{\tau, \tau'\} \cdot \min_{lex}\{\eta, \eta'\}$. For that reason, it is save always only to store the lexicographic minimum. Furthermore, for determining the lexicographic minimum for going from s to q on $b_1 b_2 \ldots b_m$, it is sufficient to know for each state p the lexicographic minimum of the history on the first n steps and then the history for the resulting m steps will be one of these strings in $\{0,1\}^{n+1}$.

The disadvantage of this approach is that the information archived is growing and growing, that is, in each step the strings archived in the pairs get longer and longer. Therefore, one does the more complicated following algorithm to update the state, which is represented by a set of pairs (p, τ) where all the strings in the state are of same length and where for each $p \in Q$ there is at most one pair (p, τ) in the state of the Muller automaton. Furthermore, one stores in the state besides these pairs one special symbol representing a number between 0 and the maximum length of a τ which represents a column deleted in the last step. For the next state, this number is irrelevant.

First one creates the initial state $\tilde{s}$ of the Muller automaton by taking $\{\infty, (s, 0)\}$ in the case that $s \in F$ and $\{\infty, (s, 1)\}$ in the case that $s \notin F$. Now, one determines for each state $\tilde{p}$ created so far and each symbol b a successor state $\delta(p)$ as follows; one keeps adding these successor states to the set P of states of the Muller automaton until no new state is created this way. Let a state $\tilde{p} \in P$ and a symbol $b \in \Sigma$ be given.

1. Let $Q' = \{q : \exists \sigma_q \, [(q, \sigma_q) \in \tilde{p}]\}$.
2. Now one determines whether there is a k such that for all σ_q, σ_p: The first bit differing in σ_p, σ_q is not at position k and if $\sigma_p(k) = 0$ then there is a $k' > k$ with $\sigma_p(k') = 0$ as well. If this k exists then choose the least among all possible values else let $k = \infty$.
3. Start creating the state $\Delta(\tilde{p}, b)$ by putting the element k into this state (which is considered to be different from all pairs).
4. Let τ_q be obtained from σ_q by omitting the bit at position q in the case $k < \infty$ and by letting $\tau_q = \sigma_q$ in the case $k = \infty$.
5. For each $q \in F$, determine the set $\{\tau_p 0 : p \in Q' \wedge (p, b, q) \in \delta\}$. If this set is not empty, then let η_q is lexicographic minimum and put (q, η_q) into the new state $\Delta(\tilde{p}, b)$.
6. For each $q \in Q - F$, determine the set $\{\tau_p 1 : p \in Q' \wedge (p, b, q) \in \delta\}$. If this set is not empty, then let η_q is lexicographic minimum and put (q, η_q) into the new state $\Delta(\tilde{p}, b)$.
7. The new state $\Delta(\tilde{p}, b)$ consists of all the information put into this set by the above algorithm.

Now one shows that whenever the σ_q in the state $\tilde{p}$ have at least length $2 * |Q|$ then $k < \infty$ and therefore the η_q in the state $\Delta(\tilde{p}, b)$ will have the same length as the σ_q, hence the length of the archived strings is not increasing and so the number of states created is finite, actually P has at most $2^{2*|Q|^2+|Q|} * (2 * |Q| + 1)^2$ members.

To see this, assume that all the strings σ_q in $\tilde{p}$ have length $2 * |Q|$. There are at most $|Q|$ of these strings. They and their prefixes form a binary tree with up to $|Q|$ leaves of length $2*|Q|$ and so there are at most $|Q|-1$ branching nodes σ' in the tree. For each branching node σ', there are $\sigma'0, \sigma'1$ in the tree. Now let $K = \{|\sigma'| : \sigma'$ is a branching node in the tree$\}$. Furthermore, for each leave σ_p, let k' be the largest value where $\sigma_p(k') = 0$; for each σ_p add this k' into K. Then K has at most $2*|Q|-1$ many members. Hence $k = \min(\{0, 1, \ldots, 2 * |Q| - 1\} - K)$ exists and is identical to the k chosen in step 2, as for all σ_p, σ_q, if $\sigma_p(k) = 0$ then some $\sigma_p(k') = 0$ for $k' > k$ and if $\sigma_p(k) \neq \sigma_q(k)$ then there is some $k' < k$ with $\sigma_p(k') \neq \sigma_q(k')$. Hence the τ_q are shorter than the σ_q by one bit and the η_q have the same length as the τ_q. Furthermore, it is clear that Δ is a function and the resulting Muller automaton will be deterministic.

The remaining part of the Muller automaton to be defined is G: So let G contain every set W such that there is a $k < \infty$ satisfying $k = \min\{k' : \exists \tilde{p} \in W\, [k' \in \tilde{p}]\}$ and $W \cap U_k \neq \emptyset$ where

$$U_k = \{\tilde{p} : \exists \vartheta \in \{0,1\}^k\, [\exists (q,\sigma) \in \tilde{p}\, [\sigma \in \vartheta \cdot \{0,1\}^*] \wedge \forall (q,\sigma) \in \tilde{p}\, [\sigma \notin \vartheta \cdot 1^*]]\}.$$

Consider any given ω-word and let W be the set of states which the Muller automaton visits on this ω-word infinitely often. Let $\tilde{p}_m$ denote $\Delta(\tilde{s}, b_1 b_2 \ldots b_m)$. There is an n so large that $\tilde{p}_m \in W$ for all $m \geq n$.

First assume that there is an $\tilde{p}_{n'} \in U_k$ for some $n' > n$. Now the Muller automaton accepts $b_1 b_2 \ldots$, as $p_{n'} \in W$. So one has to show that $b_1 b_2 \ldots$ is also accepted by the Büchi automaton. As $\tilde{p}_{n'} \in U_k$ there is a $\vartheta \in \{0,1\}^k$ satisfying $\exists (q,\sigma) \in \tilde{p}_{n'}\, [\sigma \in \vartheta \cdot \{0,1\}^*] \wedge \forall (q,\sigma) \in \tilde{p}_{n'}\, [\sigma \notin \vartheta \cdot 1^*]$. Now consider $\Delta(\tilde{p}_{n'}, b_{n'})$. By construction, whenever $(q,\sigma) \in \tilde{p}_{n'}$ and σ extends ϑ then $\sigma(k') = 0$ for some $k' \geq k$; this property is preserved by the corresponding τ associated with q in $\tilde{p}_{n'}$. Furthermore, each pair $(q',\sigma) \in \tilde{p}_{n'+1}$ satisfies that $\sigma = \tau a$ for some a and the lexicographically least τ which belongs to some $q \in Q$ with $(q, b_{n'}, q') \in \delta$; whenever that τ extends ϑ then it contains a 0 after position k and therefore η has the same property. Hence every pair $(q',\sigma) \in \tilde{p}_{n'+1}$ satisfies that either σ does not extend ϑ or σ extends ϑ with a string containing a 0. Furthermore, if some of the $\tilde{p}_m$ with $m > n'$ would not contain any (q, σ_q) with σ_q extending ϑ, then this property would inherit to all $\tilde{p}_o$ with $o > m$; as $\tilde{p}_{n'} = \tilde{p}_o$ for infinitely many o, this cannot happen. Hence all members of W are in U_k as witnessed by the same ϑ.

Now let T be the tree of all finite runs $q_0 q_1 \ldots q_m$ such that there is an associated sequence $(\sigma_0, \sigma_1, \ldots, \sigma_m)$ of strings with $(q_h, \sigma_h) \in \tilde{p}_h$ for all $h \leq m$ and $\vartheta \preceq \sigma_h$ for all h with $n \leq h \leq m$ and satisfying for all $h < m$ and the $k' \in \tilde{p}_{h+1}$ that $(q_h, b_h, q_{h+1}) \in \delta$ and σ_{h+1} is obtained from σ_h by omitting the k'-th bit (in the case that $k' \neq \infty$) and then appending 0 in the case that $q_{h+1} \in F$ and appending 1 in the case that $q_{h+1} \notin F$. Each pair in each $\tilde{p}_m$ for each m must be reachable by such a sequence; hence T is infinite. By König's Lemma there is an infinite sequence of q_m of states with the corresponding sequence of σ_m with the same property. This sequence then satisfies that from some n onwards there is always a 0 somewhere after the k-th bit in σ_m; furthermore, the k-th bit is infinitely often deleted; hence it is needed that infinitely often a 0 gets appended and so the sequence $q_0 q_1 \ldots$ satisfies that $q_m \in F$ for infinitely many m. Hence $b_0 b_1 \ldots$ has the accepting run $q_0 q_1 \ldots$ of the Büchi automaton.

Second assume that there is no $\tilde{p}_m \in U_k$ for any $m \geq n$. Then the Muller automaton rejects the run and one has to show that the Büchi automaton does the same. Assume by way of contradiction that there is an accepting run $q_0 q_1 q_2 \ldots$ of the Büchi automaton on this sequence and let $\sigma_0, \sigma_1, \sigma_2, \ldots$ be the corresponding strings such that $(q_m, \sigma_m) \in \tilde{p}_m$ for all m. There is a string $\vartheta \in \{0,1\}^k$ which is for almost all m a prefix of the σ_m. Furthermore, by assumption, there is for all $m \geq n$ a state $r_m \in Q$ with $(r_m, \eta 1^\ell) \in \tilde{p}_m$. There are infinitely many m with $\sigma_m \succeq \vartheta \wedge \sigma_m \neq \vartheta 1^\ell$. Let k' be the minimum of the $k' \geq k$ such that $\sigma_{n''}(k') = 0 \wedge \vartheta \preceq \sigma_{n''}$ for some $m \geq n$; the minimal k' exists; fix some n'' with the chosen property. Then the number $k'' \in \tilde{p}_{n''+1}$ satisfies that $k'' \geq k$. Furthermore, $k'' \neq k'$ as $(r_{n''}, \vartheta 1^\ell)$ and $(q_{n''}, \sigma_{n''})$ are both in $\tilde{p}_{n''}$ and k' is the first position where they differ. Furthermore, it does not happen that $k \leq k'' < k'$ as then $\sigma_{n''+1}$ would have the 0 at $k'-1$ and $k'-1 \geq k$ in contradiction to the choice of k'. Hence not k but $k'+1$ would be the limit inferior of all the numbers in $\tilde{p}_m$ with $m \geq n''$ which equals to the states in W, a contradiction. Thus such an accepting run of the Büchi automaton on $b_0 b_1 b_2 \ldots$ does not exists and the Büchi automaton rejects the input in the same way as the Muller automaton does.

So it follows from the case distinction that both automata recognise the same language. Furthermore, the deterministic Muller automaton constructed from the Büchi automaton has exponentially many states in the number of states of the original non-deterministic Büchi automaton.

(b)⇒(c). This holds, as every deterministic Muller automaton can by definition also be viewed as a non-deterministic one.

(c)⇒(a). Let a non-deterministic Muller automaton $(Q, \Sigma, \delta, s, G)$ be given. Let $succ(w, W)$ be a function which cycles through the set W: if $W = \{w_1, w_2, w_3\}$ then $succ(w_1, W) = w_2$, $succ(w_2, W) = w_3$ and $succ(w_3, W) = w_1$. Now let $P = Q \times Q \times (G \cup \{\emptyset\})$ is the set of states of the equivalent Büchi automaton, the alphabet Σ is unchanged, the starting state is $(s, s, \emptyset)$ and for each transition (p, a, q) and each r and each $W \in G$ with $p, q, r \in W$, put the following transitions into Δ: $((p, p, \emptyset), a, (q, q, \emptyset))$, $((p, p, \emptyset), a, (q, q, W))$, $((p, r, W), a, (q, r, W))$ in the case that $p \neq r$ and $((p, p, W), a, (q, Succ(p, W), W))$. The set F is the set of all (p, p, W) with $p \in W$ and $W \in G$ (in particular, $W \neq \emptyset$). Now it is shown that $(P, \Sigma, (s, s, \emptyset), \Delta, F)$ is a non-deterministic Büchi automaton recognising L.

Given a word recognised by the Büchi automaton, there is an accepting run which goes infinitely often through a node of the form (p, p, W) with $p \in W$. When it is in (p, p, W), the next node is of the form (q, p', W) with $p' = Succ(p, W)$ and $q \in W$, the second parameter will remain p' until the run reaches (p', p', W) from which it will transfer to a state of the form (q', p'', W) with $p'' = Succ(p', W)$. This argument shows that the run will actually go through all states of the form (q, q, W) with $q \in W$ infinitely often and that the first component of the states visited after (p, p, W) will always be a member of Q. Hence, if one takes the first components of the run, then almost all of its states are in W and all states occur in W infinitely often and it forms a run in the given Muller automaton. Hence the given ω-word is recognised by the

Muller automaton as well.

On the other hand, if one has a run $q_0 q_1 \ldots$ accepting an ω-word in the given Muller automaton and if W is the set of states visited infinitely often and if n is the position in the run from which onwards only states in W are visited, then one can translate the run of the Muller automaton into a run of the Büchi automaton as follows: $(q_0, q_0, \emptyset)$, $(q_1, q_1, \emptyset)$, $\ldots$, $(q_n, q_n, \emptyset)$, (q_{n+1}, q_{n+1}, W). For $m > n$ and the m-th state being (q_m, r_m, W) then the r_{m+1} of the next state (q_{m+1}, r_{m+1}, W) is chosen such that $r_{m+1} = Succ(W, r_m)$ in the case $q_m = r_m$ and $r_{m+1} = r_m$ in the case $q_m \neq r_m$. It can be seen that all the transitions are transitions of the Büchi automaton. Furthermore, one can see that the sequence of the r_m is not eventually constant, as for each r_m there is a $k \geq m$ with $q_k = r_m$, $r_k = r_m$ and $r_{m+1} = Succ(r_m, W)$. Hence one can conclude that the states of the form (p, p, W) with $p \in W$ are infinitely often visited in the run and the Büchi automaton has also an accepting run for the given word. Again the construction gives an exponential upper bound on the number of states of the Büchi automaton constructed from the Muller automaton. This completes the proof. ∎

In the above proof, an exponential upper bound on the number of states means that there is a polynomial f such that the number of states in the new automaton is bounded by $2^{f(|Q|)}$ where Q is the set of states of the old automaton and $|Q|$ denotes the number of states. So the algorithm gives the implicit bound that if one computes from some non-deterministic Büchi automaton another non-deterministic Büchi automaton recognising the complement then the number of states is going up in an double-exponential way, that is, there is some polynomial g such that the number of states in the Büchi automaton recognising the complement is bounded by $2^{2^{g(|Q|)}}$. This bound is not optimal, as the next result shows, but it can be improved to an exponential upper bound. Schewe [69] provides a tight exponential bound, the following theorem just takes the previous construction to give some (non-optimal) way to complement a Büchi automaton which still satisfies an exponential bound.

Theorem 10.10. *Assume that $(Q, \Sigma, \delta, s, F)$ is a non-deterministic Büchi automaton recognising the language L. Then there is an only exponentially larger automaton recognising the complement of L.*

Proof. For the given automaton for a language L, take the construction from Theorem 10.9 (a)⇒(b) to find a deterministic Muller automaton for L with a set P of states, a transition function Δ, $\tilde{p}$ and the numbers k and sets U_k defined as there. Now define the new state-space as $R = P \cup \{\tilde{p} \cup \{\hat{h}\} : \exists k\, [h \leq k < \infty \wedge k \in \tilde{p}]\}$ where $\hat{0}, \hat{1}, \ldots$ are considered as different from $0, 1, \ldots$ (for avoiding multi-sets) and the mapping $h \mapsto \hat{h}$ is one-one. So besides the states in P, there are states with an additional number $\hat{h}$ which is considered as a commitment that the value of the numbers in future states will never be below h. Note that $|R| \leq (2 * |Q| + 1) * |P|$ so that the exponential bound on R is only slightly larger than the one on P. The idea is that the transition relation on R follows in general Δ with the additional constraints, that at any time a commitment can be made but it can never be revised; furthermore, the commitment cannot be violated. So the following transitions are possible:

1. $(\tilde{p}, a, \Delta(\tilde{p}, a))$ if $\tilde{p} \in P$;
2. $(\tilde{p}, a, \Delta(\tilde{p}, a) \cup \{\hat{h}\})$ if $\tilde{p} \in P$ and h is any number between 0 and $2 * |Q|$;
3. $(\tilde{p} \cup \{\hat{h}\}, a, \Delta(\tilde{p}, a) \cup \{\hat{h}\})$ if $\tilde{p} \in P$ and there is a $h' \geq h$ with $h' \in \Delta(\tilde{p})$.

The set F has the role to enforce that for the limit inferior k of the numbers occurring in the states $\tilde{p}$, the commitment $\hat{k}$ is made eventually and some state $\tilde{p} \cup \{\hat{k}\}$ is visited infinitely often with $k \in \tilde{p} \wedge \tilde{p} \notin U_k$. Note that Theorem 10.9 (a)$\Rightarrow$(b) showed that this is exactly the behaviour of the underlying Muller automaton (without the commitments) when running on an ω-word: it rejects this ω-word iff it runs through a state $\tilde{p}$ infinitely often such that $k \in \tilde{p}$ for the limit inferior k of the numbers encoded into each of the infinitely often visited states and $\tilde{p} \notin U_k$. Indeed, all the states visited infinitely often are either all in U_k or all outside U_k. Therefore, one can choose F as follows:

$$F = \{\tilde{p} \cup \{\hat{k}\} : \tilde{p} \in P \wedge k \in \tilde{p} \wedge \tilde{p} \notin U_k\}.$$

Now the non-deterministic part of this Büchi automaton is to eventually guess k and do the corresponding commitment; if it then goes infinitely often through a node $\tilde{p} \cup \{\hat{k}\}$ of F, then the fact that $k \in \tilde{p}$ enforces that the limit inferior of the positions deleted in the strings is k; furthermore, the commitment enforces that it is not below k. Hence the simulated Muller automaton has the parameter k for its limit and cycles infinitely often through a node not in U_k; this implies that it rejects the ω-word and that the word is not in L. It is however accepted by the new Büchi automaton.

On the other hand, if the new Büchi automaton has only rejecting runs when chosing the right parameter k, then therefore all the states through which the Büchi automaton cycles through an infinite run with the right commitment are of the form $\tilde{p} \cup \{\tilde{k}\}$ with $\tilde{p} \in U_k$; hence the underlying Muller automaton accepts the ω-word and the Büchi automaton correctly rejects the ω-word. Hence the new Büchi automaton recognises the complement of L. ∎

Exercise 10.11. *Let $\Sigma = \{0, 1, 2\}$ and a parameter h be given. Make a non-deterministic Büchi automaton recognising the language L of all ω-words $b_1 b_2 \ldots$ in which there are infinitely many m such that $b_m = b_{m+h}$. Give a bound on the number of states of this automaton. Construct a Muller automaton recognising the same language and a Büchi automaton recognising the complement of L. How many states do these automata have (in terms of the value h)?*

Description 10.12: Rabin and Streett Automata. Rabin and Streett automata are automata of the form $(Q, \Sigma, \delta, s, \Omega)$ where Ω is a set of pairs (E, F) of subsets of Q and a run on an ω-word $b_0 b_1 b_2 \ldots$ is a sequence $q_0 q_1 q_2 \ldots$ with $q_0 = s$ and $(q_n, b_n, q_{n+1}) \in \delta$ for all n; such a run is accepting iff the set $U = \{p \in Q : \exists^\infty n\, [p = q_n]\}$ of infinitely often visited nodes satisfies

- in the case of Rabin automata that $U \cap E \neq \emptyset$ and $U \cap F = \emptyset$ for one pair $(E, F) \in \Omega$;
- in the case of Streett automata that $U \cap E \neq \emptyset$ or $U \cap F = \emptyset$ for all pairs $(E, F) \in \Omega$.

Given a deterministic Rabin automaton $(Q, \Sigma, \delta, s, \Omega)$, the Streett automaton

$$(Q, \Sigma, \delta, s, \{(F, E) : (E, F) \in \Omega\})$$

recognises the complement of the Rabin automaton.

Example 10.13. Assume that an automaton with states $Q = \{q_0, q_1, \ldots, q_9\}$ on seeing digit d goes into state q_d. Then the condition Ω consisting of all pairs $(Q - \{q_d\}, \{q_d\})$ produces an Rabin automaton which accepts iff some digit d appears only finitely often in a given ω-word.

Assume that an automaton with states $Q = \{s, q_0, q_1, \ldots, q_9\}$, start state s and transition function δ given by $\delta(s, d) = q_d$ and if $d = e$ then $\delta(q_d, e) = d$ else $\delta(q_d, e) = s$. Let $E = \{q_0, q_1, \ldots, q_9\}$ and $F = \{s\}$ and $\Omega = \{(E, F)\}$. This automaton is a deterministic Rabin automaton which accepts all ω-words where exactly one digit occurs infinitely often.

Furthermore, if one takes $\Omega = \{(\emptyset, \{s\})\}$ then one obtains a Streett automaton which accepts exactly the ω-words where exactly one digit occurs infinitely often, as the automaton cannot obtain that a state in $\emptyset$ comes up infinitely often and therefore has to avoid that the state s is visited infinitely often. This happens exactly when one digit comes infinitely often.

Quiz 10.14. *Give an algorithm to translate a Büchi automaton into a Streett automaton.*

Exercise 10.15. *Assume that, for $k = 1, 2$, an ω-language L_k is recognised by a Streett automaton $(Q_k, \Sigma, s_k, \delta_k, \Omega_k)$. Prove that then there is a Streett automaton recognising $L_1 \cap L_2$ with states $Q_1 \times Q_2$, start state (s_1, s_2), transition relation $\delta_1 \times \delta_2$ and an Ω containing $|\Omega_1| + |\Omega_2|$ pairs. Here $(\delta_1 \times \delta_2)((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$. Explain how Ω is constructed.*

Selftest 10.16. *Assume that a finite game has the nodes* $\{00, 01, \ldots, 99\}$ *and the game can go from ab to bc for all* $a, b, c \in \{0, 1, \ldots, 9\}$*; except for the node* 00 *which is the target node. The player reaching* 00 *first wins the game. Which nodes are winning the first player which reaches* 0 *first wins. Which nodes are winning nodes for Anke and which are winning nodes for Boris and which are draw nodes?*

Selftest 10.17. *Consider a game on* $\mathbb{N}^3$ *where each player can move from* (a, b, c) *to* (a', b', c') *if either* $a = a' \wedge b = b' \wedge c = c' + 1$ *or* $a = a' \wedge b = b' + 1 \wedge c = 0$ *or* $a = a' + 1 \wedge b = 0 \wedge c = 0$*. The player which moves into* $(0, 0, 0)$ *loses the game, so other than in the previous task, each player has to try to avoid moving to* $(0, 0, 0)$*. Are there draw nodes in the game? Is every play of the game finite? The node* $(0, 0, 0)$ *is not counted.*

Selftest 10.18. *Let* $V = \{0, 1, \ldots, 19\}$ *and* $E = \{(p, q) : q \in \{p + 1, p + 2, p + 3\}$ $($*modulo* $20)\}$*; the starting node is* 0*. Does Anke have a memoryless winning strategy for the set* $\{6, 16\}$ *of nodes which must be visited infinitely often? If so, list out this memoryless winning strategy, if not, say why it does not exist.*

Selftest 10.19. *Explain the differences between a Büchi game, a survival game and an update game.*

Selftest 10.20. *Construct a non-deterministic Büchi automaton recognising the* ω*-language* $\{0, 1, 2\}^* \cdot \{001, 002\}^\omega \cup \{0, 1, 2\}^* \cdot \{01, 02\}^\omega$.

Selftest 10.21. *Construct a deterministic Büchi automaton which recognises the* ω*-language* $\{00, 11\}^* \cdot \{11, 22\}^* \cdot \{22, 00\}^\omega$.

Selftest 10.22. *Prove that if a Rabin automaton recognises an* ω*-language* L *so does a Büchi automaton.*

Solution for Selftest 10.16. *The winning nodes for Anke are* $\{10, 20, \ldots, 90\}$ *where she moves to* 00*; for all other nodes, Anke must avoid that Boris can move to* 00 *and would move from* ab *to any number* bc *with* $c \neq 0$*. Boris does the same, hence all numbers of the form* ab *with* $b \neq 0$ *are draw nodes.*

Solution for Selftest 10.17. *One can see that every move of the game from* (a, b, c) *to* (a', b', c') *satisfies* $(a', b', c') <_{lex} (a, b, c)$ *for the lexicographic order on triples of numbers. As this ordering is, on* $\mathbb{N}^3$*, a well ordering, every play of the game must eventually go to* $(0, 0, 0)$ *and one of the players loses and the other one wins. So there are no infinite plays and thus also no draw nodes from which both players could enforce an infinite play.*

Solution for Selftest 10.18. *Yes, Anke has a memoryless winning-strategy and it is given by the following table:* $0 \to 3$, $1 \to 3$, $2 \to 3$, $3 \to 6$, $4 \to 6$, $5 \to 6$, $6 \to 7$, $7 \to 8$, $8 \to 9$, $9 \to 10$, $10 \to 13$, $11 \to 13$, $12 \to 13$, $13 \to 16$, $14 \to 16$, $15 \to 16$, $16 \to 17$, $17 \to 18$, $18 \to 19$, $19 \to 0$*. When the game is in node* p *with* $0 \leq p \leq 2$ *then it goes to Anke either in* p *or in some node* q *with* $p+1 \leq q \leq p+3$*. If it goes to Anke in node* p *then she moves to* 3*; now Boris can either move to* 6 *and satisfy the visit-requirement or to* 4 *or* 5 *in which case Anke satisfies the visit-requirement for* 6 *in the next move. If it is Boris move while the game is in* 0 *or* 1 *or* 2*, he can move it to* 1, 2, 3, 4 *or* 5 *and either Anke will move to* 3 *or to* 6 *directly, if the game goes to* 3 *it will also eventually visit* 6*. Similarly one shows when the game comes across some of the nodes* $10, 11, 12$ *it will eventually go to* 16 *with Anke's winning strategy. Thus Anke can play a memoryless winning strategy which enforces infinitely many visits of* 6 *and* 16 *in this update game.*

Solution for Selftest 10.19. *A survival game is a game where Anke wins if the game runs forever. A Büchi game has in addition to the set* V *of nodes also a subset* W *such that Anke wins iff the game runs forever and at least one node of* W *is visited infinitely often. Every survival game is also a Büchi game (by chosing* $W = V$*) but not vice versa. An update game is a game which has like a Büchi game a selected set* W *of nodes; the difference to the Büchi game is, however, that every node in* W *must be visited infinitely often. While Büchi games have always memoryless winning strategies, update games might fail to do so (at least for player Anke).*

Solution for Selftest 10.20. *The Büchi automaton has states* $q_0, q_1, q_2, q_3, q_4, q_5$ *and* q_0 *is the start state,* q_1, q_3 *are the accepting states and one can go on* $0, 1, 2$ *from* q_0 *to* q_0, q_1, q_3*, from* q_1 *on* 0 *to* q_2 *and from* q_2 *on* $1, 2$ *to* q_1*, from* q_3 *on* 0 *to* q_4*, from* q_4 *on* 0 *to* q_5 *and from* q_5 *on* $1, 2$ *to* q_3*. Thus when going to* q_1 *the automaton only will process further inputs from* $\{01, 02\}^\omega$ *and when going to* q_3*, the automaton will only process further inputs from* $\{001, 002\}^\omega$*.*

Solution for Selftest 10.21. *The table of the Büchi automaton looks as follows:*

state	*type*	0	1	2
q_0	*start,reject*	$q_{0,0}$	$q_{0,1}$	$q_{1,2}$
$q_{0,0}$	*reject*	q_0	–	–
$q_{0,1}$	*reject*	–	q_1	–
q_1	*reject*	$q_{2,0}$	$q_{1,1}$	$q_{1,2}$
$q_{1,1}$	*reject*	–	q_1	–
$q_{1,2}$	*reject*	–	–	q_1
q_2	*accept*	$q_{2,0}$	–	$q_{2,2}$
$q_{2,0}$	*reject*	q_2	–	–
$q_{2,2}$	*reject*	–	–	q_2

It accepts an infinite word iff it goes infinitely often through q_2. After the first time it went thorough this node, it will only process concatenations of 00 *and* 22*, as required.*

Solution for Selftest 10.22. *The idea is that for every e, E, F with $(E, F) \in \Omega$ and $e \in E$, one considers the regular language A_e of all words w such that the given Rabin automaton can go on w from the start state to e and the language $B_{e,F}$ of all non-empty words v such that the Rabin automaton can go from e to e on v without visiting any state in F. Now the automaton can on every ω-word from $A_e \cdot B_{e,F}^\omega$ go in the A_e-part of the ω-word from the start state to e and then in the $B_{e,F}^\omega$-part of the ω-word cycle from e to e without visiting any state from F. Thus there is an infinite run on the ω-word where the state e from E is visited infinitely often while no state from F is visited infinitely often and so $A_e \cdot B_{e,F}$ is a subset of the language recognised by the Rabin automaton. One can see that the language of all ω-words recognised by the Rabin automaton is the union of all $A_e \cdot B_{e,F}^\omega$ for which there is an E with $e \in E$ and $(E, F) \in \Omega$. So the ω-language recognised by the Rabin automaton is of the form from Theorem 10.5 and therefore recognised by a Büchi automaton.*

11 Automatic Functions and Relations

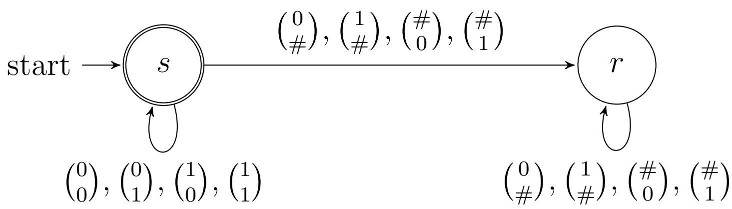
So far, only regular sets were considered. The notion of the regular sets has been generalised to automatic relations and functions.

Definition 11.1: Automatic Relations and Functions [33, 34, 45]**.** *A relation $R \subseteq X \times Y$ is automatic iff there is an automaton reading both inputs at the same speed (one symbol per cycle with a special symbol # given for inputs which are exhausted) such that $(x, y) \in R$ iff the automaton is in an accepting state after having read both, x and y, completely.*

Similarly one can define that a relation of several parameters is automatic.

A function $f : X \to Y$ is automatic iff the relation $\{(x, y) : x \in dom(f) \wedge y = f(x)\}$ is automatic.

Example 11.2. The relation $|x| = |y|$ is an automatic relation, given by an automaton which remains in an accepting starting state as long as both inputs are in Σ and transfers to a rejecting state (which it does not leave) whenever exactly one of the inputs x, y is exhausted.



Here $\binom{a}{b}$ means that the first input (x) provides an a and the second input (y) provides a b; the alphabet is $\{0, 1\}$.

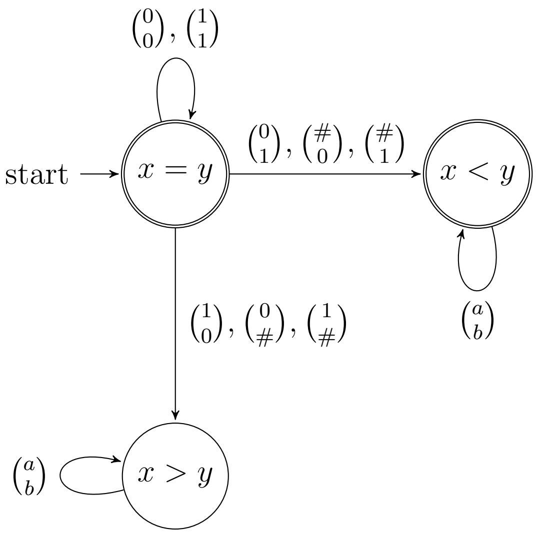
For example, if $x = 00$ and $y = 1111$, then the automaton starts in s, reads $\binom{0}{1}$ and remains in s, reads $\binom{0}{1}$ and remains in s, reads $\binom{\#}{1}$ and goes to r, reads $\binom{\#}{1}$ and remains in state r. As now both inputs are exhausted and the automaton is in a rejecting state, it rejects the input; indeed, $|00| \neq |1111|$.

If $x = 010$ and $y = 101$, then the automaton starts in s and remains in s while processing $\binom{0}{1}$, $\binom{1}{0}$ and $\binom{0}{1}$.

Notation 11.3. A set of pairs, or in general of tuples, can be directly be written in the way as the automaton would read the inputs. For this let $conv(x, y)$ be the set of pairs with symbols from x and y at the same position, where # is used to fill the shorter string (if applicable). So $conv(00, 111) = \binom{0}{1}\binom{0}{1}\binom{\#}{1}$, $conv(101, 222) = \binom{1}{2}\binom{0}{2}\binom{1}{2}$ and $conv(0123, 33) = \binom{0}{3}\binom{1}{3}\binom{2}{\#}\binom{3}{\#}$. For this, it is always understood that the symbol # is not in the alphabet used for x and y; if it would be, some other symbol has to be used for denoting the empty places of the shorter word. So a relation $R \subseteq \Sigma^* \times \Sigma^*$ is automatic iff the set $\{conv(x, y) : (x, y) \in R\}$ is regular. Similarly for relations with an arity other than two.

Example 11.4. Assume that the members of Σ are ordered; if $\Sigma = \{0, 1\}$ then the default ordering is $0 < 1$. One says that a string $v = a_1 a_2 \ldots a_n$ is lexicographic before a string $w = b_1 b_2 \ldots b_m$ iff either $n < m$ and $a_1 = b_1 \wedge a_2 = b_2 \wedge \ldots \wedge a_n = b_n$ or there is a $k < \min\{n, m\}$ with $a_1 = b_1 \wedge a_2 = b_2 \wedge \ldots \wedge a_k = b_k \wedge a_{k+1} < b_{k+1}$. One writes $v <_{lex} w$ if v is lexicographic before w; $v \leq_{lex} w$ means either $v <_{lex} w$ or $v = w$. So $000 <_{lex} 00110011 <_{lex} 0101 <_{lex} 010101 <_{lex} 1 <_{lex} 10 <_{lex} 100 <_{lex} 11$.

The lexicographic ordering is an automatic relation. For the binary alphabet $\{0, 1\}$, it is recognised by the following automaton.



Here $\binom{a}{b}$ on an arrow means that the automaton always goes this way.

Exercise 11.5. *Say in words which automatic relations are described by the following regular expressions:*

- $\{\binom{0}{0}, \binom{0}{1}, \binom{1}{0}, \binom{1}{1}\}^* \cdot \{\binom{0}{0}, \binom{1}{1}\} \cdot (\{\binom{\#}{0}, \binom{\#}{1}\}^* \cup \{\binom{0}{\#}, \binom{1}{\#}\}^*)$,
- $\{\binom{0}{1}, \binom{1}{0}\}^* \cdot (\{\varepsilon\} \cup \{\binom{2}{\#}\} \cdot \{\binom{0}{\#}, \binom{1}{\#}, \binom{2}{\#}\}^*)$,
- $\{\binom{0}{0}, \binom{1}{0}, \binom{2}{0}, \ldots, \binom{9}{0}\}^*$?

Which of these three relations define functions? What are the domains and ranges of these functions?

Exercise 11.6. *Which of the following relations are automatic (where x_k is the k-th symbol of $x = x_1 x_2 \ldots x_n$ and $|x| = n$):*

- $R_1(x, y, z) \Leftrightarrow \forall k \in \{1, 2, \ldots, \min\{|x|, |y|, |z|\}\} \, [x_k = y_k \vee x_k = z_k \vee y_k = z_k]$;
- $R_2(x, y, z) \Leftrightarrow |x| + |y| = |z|$;
- $R_3(x, z) \Leftrightarrow \exists y \, [|x| + |y| = |z|]$;
- $R_4(x, y, z) \Leftrightarrow \exists k \in \{1, 2, \ldots, \min\{|x|, |y|, |z|\}\} \, [x_k = y_k = z_k]$;
- $R_5(x, y, z) \Leftrightarrow \exists i, j, k \, [x_i = y_j = z_k]$;
- $R_6(x, y) \Leftrightarrow y = 012 \cdot x \cdot 012$.

Give a short explanations why certain relations are automatic or not; it is not needed to construct the corresponding automata by explicit tables or diagrams.

Theorem 11.7: First-Order Definable Relations [45]**.** *If a relation is first-order-definable using automatic functions and relations then it is automatic; if a function is first-order-definable using automatic functions and relations, then it is automatic. Furthermore, one can construct the automata effectively from the automata used in the parameters to define the relation or function.*

Example 11.8. The length-lexicographic or military ordering can be defined from the two previously defined orderings: $v <_{ll} w$ iff $|v| < |w|$ or $|v| = |w| \wedge v <_{lex} w$. Hence the length-lexicographic ordering is automatic.

Furthermore, for every automatic function f and any regular subset R of the domain of f, the image $f(R) = \{f(x) : x \in R\}$ is a regular set as well, as it is first-order definable using f and R as parameters:

$$y \in f(R) \Leftrightarrow \exists x \in R \, [f(x) = y].$$

Exercise 11.9. *Let I be a regular set and $\{L_e : e \in I\}$ be an automatic family, that is, a family of subsets of Σ^* such that the relation of all (e, x) with $e \in I \wedge x \in L_e$ is automatic. Note that $D = \bigcup_{i \in I} L_i$ is first-order definable by*

$$x \in D \Leftrightarrow \exists i \in I \, [x \in L_i],$$

hence D is a regular set. Show that the following relations between indices are also automatic:

- $\{(i, j) \in I \times I : L_i = L_j\}$;

- $\{(i,j) \in I \times I : L_i \subseteq L_j\}$;
- $\{(i,j) \in I \times I : L_i \cap L_j = \emptyset\}$;
- $\{(i,j) \in I \times I : L_i \cap L_j$ *is infinite*$\}$.

Show this by showing that the corresponding relations are first-order definable from given automatic relations. One can use for the fourth the length-lexicographic order in the first-order definition.

Example 11.10. Let (N, Σ, P, S) be a grammar and $R = \{(x,y) \in (N \cup \Sigma)^* \times (N \cup \Sigma)^* : x \Rightarrow y\}$ be the set of all pairs of words where y can be derived from x in one step. The relation R is automatic.

Furthermore, for each fixed n, the relation $\{(x,y) : \exists z_0, z_1, \ldots, z_n\, [x = z_0 \wedge y = z_n \wedge z_0 \Rightarrow z_1 \wedge z_1 \Rightarrow z_2 \wedge \ldots \wedge z_{n-1} \Rightarrow z_n]\}$ of all pairs of words such that y can be derived from x in exactly n steps is automatic.

Similarly, the relation of all (x,y) such that y can be derived from x in at most n steps is automatic.

Remark 11.11. In general, the relation $\Rightarrow^*$ is not automatic for a non-regular grammar, even if the language generated by the grammar itself is regular. For example, one could consider the grammar $(\{S\}, \{0,1,2\}, \{S \to SS|0|1|2\}, S)$ generating all non-empty words over $\{0,1,2\}$. Then consider a derivation $S \Rightarrow^* S01^m2S \Rightarrow^* 0^k1^m2^n$. If $\Rightarrow^*$ would be automatic, so would be the relation of all pairs of the form $(S01^m2S, 0^k1^m2^n)$ with $k > 1 \wedge m > 0 \wedge n > 1$; this is the set of those pairs in $\Rightarrow^*$ where the first component is of the form $S011^*2S$. If the set of the convoluted pairs in $\Rightarrow^*$ is regular, so is this set.

Now, choose $n = h+4$, $m = h$, $k = h+4$ for a h much larger than the pumping constant of the assumed regular set; then the regular set of the convoluted pairs in the relation would contain for every r the string

$$\binom{S}{0}\binom{0}{0}\binom{1}{0}^{c}\binom{1}{0}^{dr}\binom{1}{0}^{h-c-d}\binom{2}{0}\binom{S}{0}\binom{\#}{1}^{h}\binom{\#}{2}^{h+4};$$

where $c \geq 0$, $d > 0$ and $h - c - d \geq 0$. In contrast to this, the condition on $\Rightarrow^*$ implies that the first S is transformed in a sequence of 0 and the second S into a sequence of 2 while the number of 1 is preserved; therefore the number $c + dr + h - c - d$ must be equal to h, which gives a contradiction for $r \neq 1$. Hence the relation $\Rightarrow^*$ cannot be automatic.

It should however be noted, that the relation $\Rightarrow^*$ is regular in the case that the grammar used is right-linear. In this case, for N denoting the non-terminal and Σ

the terminal alphabet and for every $A, B \in N$, let $L_{A,B}$ denote the set of all words w such that $A \Rightarrow^* wB$ and L_A be the set of all words w such that $A \Rightarrow^* w$. All sets L_A and $L_{A,B}$ are regular and now $x \Rightarrow^* y$ iff either $x = y$ or $x = vA$ and $y = vwB$ with $w \in L_{A,B}$ or $x = vA$ and $y = vw$ with $w \in L_A$ for some $A, B \in N$; hence the convoluted pairs of the relation $\Rightarrow^*$ form the union

$$\{conv(x,y) : x \Rightarrow^* y\} = \bigcup_{A,B \in N} (\{conv(vA, vwB) : v \in \Sigma^*, w \in L_{A,B}\} \ \cup \\ \{conv(vA, vw) : v \in \Sigma^*, w \in L_A\})$$

which is a regular set.

Exercise 11.12. *Let R be an automatic relation over $\Sigma^* \cup \Gamma^*$ such that whenever $(v, w) \in R$ then $|v| \leq |w|$ and let L be the set of all words $x \in \Sigma^*$ for which there exists a sequence $y_0, y_1, \ldots, y_m \in \Gamma^*$ with $y_0 = \varepsilon$, $(y_k, y_{k+1}) \in R$ for all $k < m$ and $(y_m, x) \in R$. Note that $\varepsilon \in L$ iff $(\varepsilon, \varepsilon) \in R$. Show that L is context-sensitive.*

Note that the converse direction of the statement in the exercise is also true. So assume that L is context-sensitive. Then one can take a grammar for $L - \{\varepsilon\}$ where each rule $v \to w$ satisfies $|v| \leq |w|$ and either $v, w \in N^+$ or $|v| = |w| \wedge v \in N^+ \wedge w \in \Sigma^+$. Now let $(x, y) \in R$ if either $x, y \in N^+ \wedge x \Rightarrow y$ or $x \in N^+ \wedge y \in \Sigma^+ \wedge (x \Rightarrow^* y$ by rules making non-terminals to terminals) or $(x, y) = (\varepsilon, \varepsilon) \wedge \varepsilon \in L$.

Theorem 11.13: Immerman and Szelepcsényi's Nondeterministic Counting [36, 72]. *The complement of a context-sensitive language is context-sensitive.*

Proof. The basic idea is the following: Given a word x of length n and a context-sensitive grammar (N, Σ, P, S) generating the language L, there is either a derivation of x without repetition or there is no derivation at all. One can use words over $\Sigma \cup N$ to represent the counter values for measuring the length of the derivation as well as the number of counted values; let u be the largest possible counter value. Now one determines using non-determinism for each ℓ how many words can be derived with derivations up to length ℓ; the main idea is that one can obtain the number for $\ell + 1$ from the number for ℓ and that the number for $\ell = 0$ is 1 (namely the start symbol S). The idea is to implement a basic algorithm is the following:

> Choose an $x \in \Sigma^*$ and verify that $x \notin L$ as follows;
> Let u be the length-lexicographically largest string in $(N \cup \Sigma)^{|x|}$;
> Let $i = Succ_{ll}(\varepsilon)$;
> For $\ell = \varepsilon$ to u Do Begin

Let $j = \varepsilon$;
For all $y \leq_{ll} u$ Do Begin

Derive the words $w_1, w_2, \ldots, w_i$ non-deterministically in length-lexicographic order in up to ℓ steps each and do the following checks:
If there is a w_m with $w_m \Rightarrow y$ or $w_m = y$ then let $j = Succ_{ll}(j)$;
If there is a w_m with $w_m = x$ or $w_m \Rightarrow x$ then abort the computation End;

Let $i = j$; End;

If the algorithm has not yet aborted then generate x;

This algorithm can be made more specific by carrying over the parts which use plain variables and replacing the indexed variables and meta-steps. For this, the new variables v, w to run over the words and k to count the derivation length; h is used to count the words processed so far. Recall that the special case of generating or not generating the empty word is ignored, as the corresponding entry can be patched easily in the resulting relation R, into which the algorithm will be translated.

1: Choose an $x \in \Sigma^+$ and initial all other variables as ε;

2: Let $u = (\max_{ll}(N \cup \Sigma))^{|x|}$;

3: Let $i = Succ_{ll}(\varepsilon)$ and $\ell = \varepsilon$;

4: While $\ell <_{ll} u$ Do Begin

5: Let $j = \varepsilon$;

6: Let $y = \varepsilon$;

7: While $y <_{ll} u$ Do Begin

8: Let $y = Succ_{ll}(y)$;

9: Let $h = \varepsilon$ and $w = \varepsilon$;

10: While $h <_{ll} i$ Do Begin

11: Nondeterministically replace w by w' with $w <_{ll} w' \leq_{ll} u$;

12: Let $v = S$;

13: Let $k = \varepsilon$;

14: While $(v \neq w) \wedge (k <_{ll} \ell)$ Do Begin

15: Nondeterministically replace (k, v) by (k', v') with $k <_{ll} k'$ and $v \Rightarrow v'$ End;

16: If $v \neq w$ Then abort the computation (as it is spoiled);
17: If $w = x$ or $w \Rightarrow x$ Then abort the computation (as $x \in L$);
18: If $w \neq y$ and $w \not\Rightarrow y$
19: Then let $h = Succ_{ll}(h)$;
20: Else let $h = i$;
21: End (of While Loop in 10);
22: If $w = y$ or $w \Rightarrow y$ Then $j = Succ_{ll}(j)$;
23: End (of While Loop in 7);
24: Let $i = j$;
25: Let $\ell = Succ_{ll}(\ell)$ End (of While Loop in 4);
26: If the algorithm has not yet aborted Then generate x;

The line numbers in this algorithm are for reference and will be used when making the relation R. Note that the algorithm is non-deterministic and that x is generated iff some non-deterministic path through the algorithm generates it. Pathes which lose their control information are just aborted so that they do not generate false data.

The variables used in the algorithm are $h, i, j, k, \ell, u, v, w, x, y$ whose values range over $(N \cup \Sigma)^*$ and furthermore, one uses a for the line numbers of the algorithm where a takes one-symbol words representing line numbers from the set $\{4, 7, 10, 14, 16, 18, 22, 24, 26\}$. Now the relation R is binary and connects words in Σ^* with intermediate non-terminal words represented by convolutions of the form $conv(a, h, i, j, k, \ell, u, v, w, x, y)$. The relation contains all the pairs explicitly put into R according to the following list, provided that the "where-condition" is satisfied. The arrow "$\rightarrow$" indicates the order of the two components of each pair.

Input: $\varepsilon \rightarrow conv(4, h, i, j, k, \ell, u, v, w, x, y)$ where $h = \varepsilon$, $i = Succ_{ll}(\varepsilon)$, $j = \varepsilon$, $k = \varepsilon$, $\ell = \varepsilon$, $u \in (\max(N \cup \Sigma))^+$, $v = \varepsilon$, $w = \varepsilon$, $x \in \Sigma^{|u|}$, $y = \varepsilon$;

4: Put $conv(4, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(26, h, i, j, k, \ell, u, v, w, x, y)$ into R where $\ell = u$ (for the case where the while-loop terminates);
Put $conv(4, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(7, h, i, \varepsilon, k, Succ_{ll}(\ell), u, v, w, x, \varepsilon)$ into R where $\ell <_{ll} u$ (for the case there another round of the loop is started with resetting j and y in lines 5 and 6 and then continuing in line 7);

7: Put $conv(7, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(24, h, i, j, k, \ell, u, v, w, x, y)$ into R where $y = u$ (for the case where the while-loop terminates);
Put $conv(7, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(10, \varepsilon, i, j, k, \ell, u, v, \varepsilon, x, Succ_{ll}(y))$ into R (for the case that the while-loop starts and y, h, w are updated in lines 8,9);

10: Put $conv(10, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(22, h, i, j, k, \ell, u, v, w, x, y)$ into R where $h = i$ (for the case where the while-loop terminates);
Put $conv(10, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(14, h, i, j, \varepsilon, \ell, u, S, w', x, y)$ into R where $h <_{ll} i$ and $w <_{ll} w' \leq_{ll} u$ (for the case that the body of the loop is started and the commands in lines 11, 12 and 13 are done);

14: Put $conv(14, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(14, h, i, j, k', \ell, u, v', w, x, y)$ into R where $v \neq w$ and $k <_{ll} k' \leq_{ll} \ell$ and $v \Rightarrow v'$ (for the case that the body of the loop in line 15 is done one round);
Put $conv(14, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(18, h, i, j, k, \ell, u, v, w, x, y)$ into R where $v = w$ and $w \neq x$ and $w \not\Rightarrow x$ (for the case that the loop leaves to line 18 without the computation being aborted);

18: Put $conv(18, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(10, Succ_{ll}(h), i, j, k, \ell, u, v, w, x, y)$ into R where $w \neq y$ or $w \not\Rightarrow y$ (in the case that the program goes through the then-case);
Put $conv(18, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(10, i, i, j, k, \ell, u, v, w, x, y)$ into R where $w = y$ or $w \Rightarrow y$ (in the case that the program goes through the else-case);

22: Put $conv(22, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(7, h, i, Succ_{ll}(j), k, \ell, u, v, w, x, y)$ into R where $w = y$ or $w \Rightarrow y$ (for the case that the condition of line 22 applies before going to line 7 for the next round of the loop);
Put $conv(22, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(7, h, i, j, k, \ell, u, v, w, x, y)$ into R where $w \neq y$ and $w \not\Rightarrow y$ (for the case that the condition of line 22 does not apply and the program goes to line 7 for the next round of the loop directly);

24: Put $(22, h, i, j, k, \ell, u, v, w, x, y) \rightarrow conv(4, h, j, j, k, Succ_{ll}(\ell), u, v, w, x, y)$ into R (which reflects the changes from lines 24 and 25 when completing the body of the loop starting in line 4);

Output: Put $(26, h, i, j, k, \ell, u, v, w, x, y) \rightarrow x$ into R (which produces the output after the algorithm has verified that $x \notin L$);

Special Case: Put $\varepsilon \rightarrow \varepsilon$ into R iff $\varepsilon \notin L$.

The verification that this relation is automatic as well as the proof of Exercise 11.12 are left to the reader. ∎

Exercise 11.14. *Consider the following algorithm to generate all non-empty strings which do not have as length a power of* 2.

1. *Guess* $x \in \Sigma^+$*; Let* $y = 0$*;*
2. *If* $|x| = |y|$ *then abort;*
3. *Let* $z = y$*;*
4. *If* $|x| = |y|$ *then generate* x *and halt;*
5. *Remove last* 0 *in* z*;*
6. *Let* $y = y0$*;*
7. *If* $z = \varepsilon$ *then goto 2 else goto 4.*

Make an R *as in Exercise 11.12 choosing* Γ *such that* Γ^* *contains all strings in*

$$\{conv(line, x, y, z) : line \in \{1, 2, 3, 4, 5, 6, 7\} \wedge x \in \Sigma^+ \wedge y, z \in \{0\}^*\}$$

but no non-empty string of Σ^* *– the symbols produced by the convolution of these alphabets are assumed to be different from those in* Σ*. A pair* (v, w) *should be in* R *iff* $|v| \leq |w|$ *and one of the following conditions hold:*

1. $v = \varepsilon$ *and* w *is a possible outcome of line* 1*, that is, of the form* $conv(2, x, 0, \varepsilon)$ *for some* $x \in \Sigma^+$*;*
2. v *is the configuration of the machine before executing the line coded in* v *and* w *the configuration after executing this line where the line number in* w *is either the next one after the one in* v *or a line number to which the program has branched by a condition or unconditional goto-command (whatever is applicable);*
3. v *is the configuration in line* 4 *and* w *is the value of* x*, provided that* y *and* x *have the same length and that* x *is as long as* $conv(4, x, y, z)$*.*

Give a full list of all the pairs which can occur such that, when starting from ε *and iterating along the relation in* R*, exactly those* $x \in \Sigma^*$ *can be reached which do not have a length of a power of* 2*.*

12 Groups, Monoids and Automata Theory

There are quite numerous connections between group theory and automata theory. On one hand, concepts from group theory are used in automata theory; on the other hand, one can also use automata theory to describe certain types of groups and semigroups. First the basic definitions.

Definition 12.1: Groups and Semigroups. *Let G be a set and $\circ$ be an operation on G, that is, $\circ$ satisfies for all $x, y \in G$ that $x \circ y \in G$.*
(a) *The structure $(G, \circ)$ is called a semigroup iff $x \circ (y \circ z) = (x \circ y) \circ z$ for all $x, y, z \in G$, that is, if the operation $\circ$ on G is associative.*
(b) *The structure $(G, \circ, e)$ is called a monoid iff $e \in G$ and $(G, \circ)$ is a semigroup and e satisfies $x \circ e = e \circ x = x$ for all $x \in G$.*
(c) *The structure $(G, \circ, e)$ is called a group iff $(G, \circ, e)$ is a monoid and for each $x \in G$ there is an $y \in G$ with $x \circ y = e$.*
(d) *A semigroup $(G, \circ)$ is called finitely generated iff there is a finite subset $F \subseteq G$ such that for each $x \in G$ there are n and $y_1, y_2, \ldots, y_n \in F$ with $x = y_1 \circ y_2 \circ \ldots \circ y_n$.*
(e) *A semigroup $(G, \circ)$ is finite iff G is finite as a set.*

Example 12.2. The set $(\{1, 2, 3, \ldots\}, +)$ forms a semigroup, the neutral element 0 is not in the base set and therefore it is not a monoid. Adding 0 to the set and using $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ gives the additive monoid $(\mathbb{N}, +, 0)$ of the natural numbers. The integers $(\mathbb{Z}, +, 0)$ form a group; similarly the rationals with addition, denoted by $(\mathbb{Q}, +, 0)$. Also the multiplicative structure $(\mathbb{Q} - \{0\}, *, 1)$ of the non-zero rationals is a group.

Example 12.3. A semigroup with a neutral element only from one side does not necessarily have a neutral element from the other side, furthermore, the neutral element from one side does not need to be unique.

To see this, consider $G = \{0, 1\}$ and the operation $\circ$ given by $x \circ y = x$. This operation is associative, as $x \circ (y \circ z) = x \circ y = x$ and $(x \circ y) \circ z = x \circ z = x$. Both elements, $y = 0$ and $y = 1$, are neutral from the right side: $x \circ y = x$. On the other hand, none of these two elements is neutral from the left side, as the examples $0 \circ 1 = 0$ and $1 \circ 0 = 1$ show.

Example 12.4. Let Q be a finite set and G be the set of all functions from Q to Q. Let $f \circ g$ be the function obtained by $(f \circ g)(q) = g(f(q))$ for all $q \in Q$. Let id be the identity function ($id(q) = q$ for all $q \in Q$). Then $(G, \circ, id)$ is a finite monoid.

Let $G' = \{f \in G : f \text{ is one-one}\}$. Then $(G', \circ, id)$ is a group. The reason is that for every $f \in G'$ there is an inverse f^{-1} with $f^{-1}(f(q)) = q$ for all $q \in Q$. Indeed, $G' = \{f \in G : \exists g \in G\, [id = f \circ g]\}$.

Example 12.5. Let $(Q, \Sigma, \delta, s, F)$ be a complete dfa and G be the set of all mappings f from Q to Q and id be the identity function. Now one considers the set

$$G' = \{f \in G : \exists w \in \Sigma^* \, \forall q \in Q \, [\delta(q, w) = f(q)]\}$$

which is the monoid of those functions which are all realised as operations given by a word in Σ^*. Then $(G', \circ, id)$ is a monoid as well. Note that if f is realised by v and g is realised by w then $f \circ g$ is realised by $w \cdot v$. The identity id is realised by the empty word ε.

This monoid defines an equivalence-relation on the strings: Every function realises only one string, so one can chose for each string w the function $f_w \in G'$ realised by w. Now let $v \sim w$ iff $f_v = f_w$. This equivalence relation partitions Σ^* into finitely many equivalence classes (as G' is finite).

Furthermore, $v \sim w$ and $x \sim y$ imply $vx \sim wy$: Given any $q \in Q$, it is mapped by f_{vx} to $f_x(f_v(q))$ and by f_{wy} to $f_y(f_w(q))$. As $f_v = f_w$ and $f_x = f_y$, it follows that $f_{vx}(q) = f_{wy}(q)$. Hence $f_{vx} = f_{wy}$ and $vx \sim wy$. A relation $\sim$ with this property is called a congruence-relation on all strings.

Let L be the language recognised by the given dfa. Recall the definition $L_x = \{y : xy \in L\}$ from Theorem 2.19. Note that $x \sim y$ implies that $L_x = L_y$: Note that $f_x(s) = f_y(s)$ and therefore $\delta(s, xz) = \delta(s, yz)$ for all z; hence $z \in L_x$ iff $z \in L_y$ and $L_x = L_y$. Therefore L is the union of some equivalence classes of $\sim$.

If the dfa is the smallest-possible dfa, that is, the dfa produced in the proof of Theorem 2.19, then the monoid $(G', \circ, id)$ derived from it is called the syntactic monoid of the language L.

The syntactic monoid is a widely used tool in formal language theory. An immediate consequence of the ideas laid out in the above example is the following theorem.

Theorem 12.6. *A language is regular iff it is the finite union of equivalence classes of some congruence relation with finitely many congruence classes.*

Example 12.7. Let a dfa have the alphabet $\Sigma = \{0, 1\}$ and states $Q = \{s, s', q, q', q''\}$ and the following state transition table:

state	s	s'	q	q'	q''
successor at 0	s'	s	q	q'	q''
successor at 1	q'	q'	q'	q''	q

Now the syntactic monoid contains the transition functions f_ε (the identity), f_0, f_1, f_{11} and f_{111}. The functions f_0 and f_1 are as in the state transition diagramme and f_{11} and f_{111} are given by the following table.

state	s	s'	q	q'	q''
f_{11}	q''	q''	q''	q	q'
f_{111}	q	q	q	q'	q''

Other functions are equal to these, for example $f_{110} = f_{11}$ and $f_{0000} = f_\varepsilon$.

Definition 12.8. *Let $(G, \circ)$ be a semigroup and $w \in G^*$ be a string of elements in G. Then $el_G(w)$ denotes that element of G which is represented by w. Here $el_G(\varepsilon)$ denotes the neutral element (if it exists) and $el_G(a_1 a_2 \ldots a_n)$ the group element $a_1 \circ a_2 \circ \ldots \circ a_n$. Furthermore, if R is a set of strings over G^* such that for every $x \in G$ there is exactly one $w \in R$ with $el_G(w) = x$, then one also uses the notation $el_R(v) = w$ for every $v \in G^*$ with $el_G(v) = el_G(w)$.*

Let $F \subseteq G$ be a finite set of generators of G, that is, F is finite and satisfies for every $x \in G$ that there is a word $w \in F^*$ with $el_G(w) = x$. Now the word-problem of G asks for an algorithm which checks for two $v, w \in F^*$ whether $el_G(v) = el_G(w)$.

Definition 12.9. A language L defines the congruence $\{(v, w) : L_v = L_w\}$ and furthermore L also defines the syntactic monoid $(G_L, \circ)$ of its minimal deterministic finite automaton.

Theorem 12.10. *Every finite group $(G, \circ)$ is the syntactic monoid of a language L.*

Proof. Let $L = \{v \in G^* : el_G(v) = \varepsilon\}$ be the set of words which are equal to the neutral element.

The corresponding dfa is $(G, G, \delta, s, \{s\})$ with $\delta(a, b) = a \circ b$ for all $a, b \in G$; here the same names are used for the states and for the symbols. In order to avoid clashes with the empty word, the neutral element of G is called s for this proof; it is the start symbol and also the only accepting symbol.

For each $a \in G$, the inverse b of a satisfies that b is the unique one-letter word in G^* such that $L_{ab} = L$. Therefore, for all $a \in G$, the languages L_a are different, as two different elements $a, b \in G$ have different inverses. Thus the dfa is a minimal dfa and has a congruence $\sim$ satisfying $v \sim w$ iff $el_G(v) = el_G(w)$ iff $L_v = L_w$. Note that every word $w \in G^*$ satisfies $el_G(w) = a$ for some $a \in G$. For each $a \in G$, the function f_a belonging to G is the function which maps every $b \in G$ to $b \circ a \in G$. It is easy to see that the syntactic monoid given by the f_w with $w \in G^*$ is isomorphic to the original group $(G, \circ)$. ▮

Example 12.11. There is a finite monoid which is not of the form $(G_L, \circ)$ for any language L.

Let $(\{\varepsilon, 0, 1, 2\}, \circ)$ be the semigroup with $\varepsilon \circ \varepsilon = \varepsilon$, $a \circ \varepsilon = \varepsilon \circ a = a$ and $a \circ b = b$

for all $a, b \in \{0, 1, 2\}$.

The set $\{0, 1, 2\}$ generates the given monoid. Consider all words over $\{0, 1, 2\}^*$. Two such words v, w define the same member of the semigroup iff either both v, w are empty (and they represent ε) or both v, w end with the same digit $a \in \{0, 1, 2\}$ (and they represent a).

The monoid is the syntactic monoid of the dfa which has alphabet $\Sigma = \{0, 1, 2\}$, states $Q = \{s, 0, 1, 2\}$, start state s and transition function $\delta(q, a) = a$ for all symbols a and states q. One can easily see that after reading a word wa the automaton is in the state a and initially, after reading the empty word ε, it is in s.

Consider now any language $L \subseteq \Sigma^*$ which would be a candidate for $(G_L, \circ) = (G, \circ)$. Furthermore, assume that $L \neq \emptyset$ and $L \neq \Sigma^*$. Then $(G_L, \circ)$ must contain equivalence classes of the form $\Sigma^* \cdot \{a\}$ with $a \in \Sigma$ and one equivalence class of the form $\{\varepsilon\}$ which belongs to the neutral element of the monoid — note that there is always a neutral element, as the semigroup operation belonging to reading ε does not change any state, so the equivalence class of $\{\varepsilon\}$ will not be not be fusionated with any other one except for the case that there is only one equivalence class Σ^*. Furthermore, the equivalence classes of the form $\Sigma^* \cdot \{a\}$ are due to the fact that for each non-neutral element $b \in G$, b is idempotent and $c \circ b = b$ for all other monoid elements c.

Furthermore, if for $a, b \in \Sigma$ the condition $L(a) = L(b)$ holds then in the minimal automaton of L both a, b lead to the same state: for further $c \in \Sigma$, the transition goes to the state representing the equivalence class $\Sigma^* \cdot \{c\}$. Thus there are besides ε at most two further states in the minimal automaton of L and the corresponding syntactic monoid is one of the following three (with names $0, 1$ used for the monoid elements different from ε):

(a) $(\{\varepsilon, 0, 1\}, \circ)$ with ε being the neutral element and $a \circ 0 = 0$ and $a \circ 1 = 1$ for all group elements a;
(b) $(\{\varepsilon, 0\}, \circ)$ with ε being the neutral element and $a \circ 0 = 0$ for all group elements a;
(c) $(\{\varepsilon\}, \circ)$ with $\varepsilon \circ \varepsilon = \varepsilon$.

These three syntactic monoids are all different from the given one.

Exercise 12.12. *Determine the syntactic monoids $(G_{L_k}, \circ)$ for the following languages L_1 and L_2:*

1. $L_1 = \{0^n 1^m : n + m \leq 3\}$;

2. $L_2 = \{w : w$ *has an even number of* 0 *and an odd number of* $1\}$.

Exercise 12.13. *Determine the syntactic monoids $(G_{L_k}, \circ)$ for the following languages L_3 and L_4:*

1. $L_3 = \{00\}^* \cdot \{11\}^*$*;*

2. $L_4 = \{0,1\}^* \cdot \{00, 11\}$*.*

Quiz 12.14. *Determine the syntactic monoids* $(G_{L_5}, \circ)$ *for the languages* $L_5 = \{0\}^* \cdot \{1\}^*$*.*

Description 12.15: Automatic groups and semigroups in Thurston's framework [25]**.** A Thurston automatic semigroup is a finitely generated semigroup which is represented by a regular subset $G \subseteq F^*$ such that the following conditions hold:

- G is a regular subset of F^*;
- Each element of the semigroup has exactly one representative in G;
- For each $y \in G$ the mapping $x \mapsto el_G(xy)$ is automatic.

The second condition is a bit more strict than usual; usually one only requires that there is at least one representative per semigroup element and that the relation which checks whether two representatives are equal is automatic. Furthermore, for monoids and groups, unless noted otherwise, ε is used as the representative of the neutral element.

Furthermore, a Thurston automatic semigroup $(G, \circ, \varepsilon)$ is called Thurston bi-automatic if for each $y \in G$ both mappings $x \mapsto el_G(xy)$ and $x \mapsto el_G(yx)$ are automatic functions.

A semigroup $(G', *, \varepsilon)$ is isomorphic to $(G, \circ, \varepsilon)$ iff there is a bijective function $f : G' \to G$ with $f(\varepsilon) = \varepsilon$ and $f(v * w) = f(v) \circ v(w)$ for all $v, w \in G'$. One says that $(G', *, \varepsilon)$ has a Thurston automatic representation iff it is isomorphic to a Thurston automatic semigroup and $(G', *, \varepsilon)$ has a Thurston bi-automatic representation iff it is isomorphic to a Thurston bi-automatic semigroup.

Example 12.16. Let $F = \{\overline{a}, a\}$ and $G = a^* \cup \overline{a}^*$. Then $(G, \circ, \varepsilon)$ is a Thurston automatic group with

$$\begin{aligned}
a^n \circ a^m &= a^{n+m}; \\
\overline{a}^n \circ \overline{a}^m &= \overline{a}^{n+m}; \\
a^n \circ \overline{a}^m &= \begin{cases} a^{n-m} & \text{if } n > m; \\ \varepsilon & \text{if } n = m; \\ \overline{a}^{m-n} & \text{if } n < m; \end{cases} \\
\overline{a}^n \circ a^m &= \begin{cases} \overline{a}^{n-m} & \text{if } n > m; \\ \varepsilon & \text{if } n = m; \\ a^{m-n} & \text{if } n < m. \end{cases}
\end{aligned}$$

One can see that the multiplication is realised by an automatic function whenever one of the operands is fixed to a constant. In general, it is sufficient to show that multiplication with the elements in F is automatic.

The additive group of integers $(\mathbb{Z}, +, 0)$ is isomorphic to this Thurston automatic group; in other words, this group is a Thurston automatic representation of the integers. The group element a^n represents $+n$ (so aaa is $+3$), $\overline{a}^n$ represents $-n$ and ε represents 0. The operation $\circ$ is isomorphic to the addition $+$, for example $a^n \circ \overline{a}^m$ has as result the representative of $n - m$.

Example 12.17. Let $F = \{\overline{a}, a, b, c\}$ generate a monoid with the empty string representing the neutral element, $G = (a^* \cup \overline{a}^*) \cdot \{b, c\}^*$ and assume that the following additional rules governing the group operations: $ba = c$, $ca = ab$, $a\overline{a} = \varepsilon$, $\overline{a}a = \varepsilon$. These rules say roughly that $\overline{a}$ is inverse to a and c is an element representing ba. One can derive further rules like $ab = baa$ and $ba^{2n} = a^n b$.

Now one uses these rules see that the monoid is automatic, where $w \in G$ and n is the number of c at the beginning of w (when multiplying with a) and the number of b at the end of w (when multiplying with $\overline{a}$), the parameters m, n can be any number in $\mathbb{N}$:

$$
\begin{array}{rcl}
w \circ b & = & wb; \\
w \circ c & = & wc; \\
w \circ a & = & \begin{cases} a^{m+1}b^n & \text{if } w = a^m c^n; \\ \overline{a}^m b^n & \text{if } w = \overline{a}^{m+1} c^n; \\ vcb^n & \text{if } w = vbc^n; \end{cases} \\
w \circ \overline{a} & = & \begin{cases} a^m c^n & \text{if } w = a^{m+1} b^n; \\ \overline{a}^{m+1} c^n & \text{if } w = \overline{a}^m b^n; \\ vbc^n & \text{if } w = vcb^n; \end{cases}
\end{array}
$$

To see these rules, consider an example: $bcbb \circ \overline{a} = bbabb \circ \overline{a} = bbbaab \circ \overline{a} = bbbabaa \circ \overline{a} = bbbaba = bbcc$, so $vcb^2 \circ a = vbc^2$.

Note that multiplication from the other side is not realised by an automatic function in this representation; indeed, $b \circ a^{2n}bc = a^n bbc$ and this would involve halving the length of the part a^{2n} what is impossible. Indeed, some problem of this type occurs in every Thurston automatic representation of this monoid and the monoid does not have a Thurston bi-automatic representation.

Exercise 12.18. *Consider a monoid $G = a^* b^*$ with the additional rule that $b \circ a = b$ and neutral element ε. Show that this representation is Thurston automatic but not Thurston bi-automatic. Does the monoid have a Thurston bi-automatic representation?*

Hodgson [33, 34] as well as Khoussainov and Nerode [45] took a more general approach where they did not require that group elements are represented by strings over generators. This representation can be used for all semigroups.

Description 12.19: Automatic groups and semigroups in the framework of Hodgson, Khoussainov and Nerode [33, 34, 45]**.** A structure $(G, \circ)$ is called an automatic semigroup iff G is a regular subset of Σ^* for some finite alphabet Σ, the function $\circ : G \times G \to G$ is an automatic function and $\circ$ satisfies the law of associativity, that is, satisfies $x \circ (y \circ z) = (x \circ y) \circ z$ for all $x, y, z \in G$. An automatic monoid / group is an automatic semigroup with a neutral element / a neutral element and an inverse for every group element. A semigroup has an automatic presentation iff it is isomorphic to an automatic semigroup $(G, \circ)$.

Exercise 12.20. *In order to represent* $(\mathbb{Z}, +, 0)$*, use* $\Sigma = \{0, 1, +, -\}$ *and use* 0 *to represent the* 0 *and* $a_0 a_1 \ldots a_n+$ *with* $a_n = 1$ *and* $a_0, a_1, \ldots, a_{n-1} \in \{0, 1\}$ *to represent the positive integer* $\sum_{m \leq n} a_m \cdot 2^m$ *and* $a_0 a_1 \ldots a_n-$ *with* $a_n = 1$ *and* $a_0, a_1, \ldots, a_{n-1} \in \{0, 1\}$ *to represent the negative integer* $-\sum_{m \leq n} a_m \cdot 2^m$*. Show that now the addition on the so represented group of the integers is an automatic function* (*with two inputs*)*. Explain why numbers like* 000000001+ (256) *and* 0101+ (10) *and* 010100001+ (266) *are given with the least significant bits first and not last in this presentation.*

Exercise 12.21. *Consider again the monoid* $G = a^* b^*$ *with the additional rule that* $b \circ a = b$ *from Exercise 12.18. Now, for* $h, k, i, j \in \mathbb{N}$*,*

$$a^h b^k \circ a^i b^j = \begin{cases} a^{h+i} b^j & \text{if } k = 0; \\ a^h b^{k+j} & \text{if } k > 0; \end{cases}$$

use these rules to find an automatic presentation in the sense of Hodgson, Khoussainov and Nerode for this group.

Theorem 12.22. *The strings over a finite alphabet* Δ *with at least two symbols plus the concatenation form a semigroup which has an automatic representation in the sense of Thurston but not in the sense of Hodgson, Khoussainov and Nerode.*

Proof. First, one can easily see that Δ^* itself is a regular set in which every string over Δ has a unique representation and in which Δ is the set of generators of the corresponding semigroup; concatenation with a fixed string y is an automatic function mapping x to xy.

Assume now that some regular set $G \subseteq \Sigma^*$ represents the semigroup Δ^* and that $\circ$ represents the concatenation in this semigroup G. Now let $F \subseteq G$ be the set of representatives of Δ in G. Let $F_1 = F$ and $F_{n+1} = \{v \circ w : v, w \in F_n\}$. There is a constant c such that each word in F_1 has at most length c and that $v \circ w$ has at most length $\max\{|v|,|w|\}+c$ for all $v,w \in G$. It follows by induction that all words in F_n have at most length cn.

By induction one can see that F_1 represents the strings in Δ^1 and F_n represents the strings in Δ^{2^n}. As Δ has at least 2 elements, there are at least 2^{2^n} members in the set F_n. On the other hand, F_n has at most $\sum_{m \leq nc} |\Sigma|^m$ elements, which can — assuming that $|\Sigma| \geq 2$ — be estimated with $|\Sigma|^{nc+1}$. This gives a contradiction for large n as the upper bound $n \mapsto |\Sigma|^{nc+1}$ is exponential while the lower bound $n \mapsto 2^{2^n}$ is double-exponential. So, for large n, the lower bound is not below the upper bound. Hence the above mentioned automatic representation of the monoid of strings over a finite alphabet with at least two letters is not isomorphic to an automatic monoid. ∎

Exercise 12.23. *Assume that F is finite and has at least two elements. Let $(G,\circ,\varepsilon)$ be the free group generated by F. Show that this group is not isomorphic to an automatic group (in the sense of Hodgson, Khoussainov and Nerode); that is, this group does not have an automatic presentation.*

Example 12.24. Let a,b,c be generators of the monoid satisfying $c \circ b = c$ and $b \circ a = a$ and $a \circ c = c \circ a$ which gives the equation

$$a^i b^j c^k \circ a^{i'} b^{j'} c^{k'} = \begin{cases} a^i b^{j+j'} c^{k'} & \text{if } k=0 \wedge i'=0; \\ a^{i+i'} b^{j'} c^{k'} & \text{if } k=0 \wedge i'>0; \\ a^i b^j c^{k+k'} & \text{if } k>0 \wedge i'=0; \\ a^{i+i'} c^{k+k'} & \text{if } k>0 \wedge i'>0; \end{cases}$$

where $i,j,k,i',j',k' \in \mathbb{N}$. This monoid is automatic. One can represent the group as a convolution of three copies of $(\mathbb{N},+)$ and use above formula for $\circ$. When adding these components, one has both entries available of the input and those three of the output available. Then one can test for each entry whether they are zero or whether the sum of two entries give a third. This is sufficient to verify that the update is done according to the formula above.

However, the monoid is not Thurston automatic. Intuitively, the reason is that when multiplying with c from the front or with a from the back, the corresponding deletions of the entries for b cannot be done. The deletion is needed. Assume that i,j,k are given with j much larger than i,k. Note that $(a^{i-1} \circ b^j \circ c^k) \circ a$ and $a^i \circ c^k$ represent the aame semigroup elements, thus the last multiplication in the first expression must match the long representation of $a^{i-1} \circ b^j \circ c^k$ whose length is linear in j to the shorter representation of $a^i \circ c^k$ whose length is linear in $i+k$. During this shrinking process, the representation of c^k at the end of both words must be moved to the front. This is something what an automatic function can only do for a constant value of k but not when k is a parameter ranging over many values as it is here the case.

Similarly if one would consider multiplication with constants from the front only, then the corresponding representation would also not be Thurston automatic, as when multiplying $a^i b^j c^{k-1}$ from the front with c, a deletion operation as indicated above has to be done and the representatives of c^k have moved a lot to the front, what the automatic function cannot do.

Exercise 12.25. *Let a, b be generators of a group satisfying*

$$a^h b^k \circ a^i b^j = \begin{cases} a^{h+i} b^{k+j} & \textit{if } k \textit{ is even;} \\ a^{h-i} b^{k+j} & \textit{if } k \textit{ is odd;} \end{cases}$$

where $h, k, i, j \in \mathbb{Z}$. Show that this group is Thurston bi-automatic as well as automatic; find for both results representations.

There is no finitely generated group which is automatic in the sense of Hogdson, Khoussainov and Nerode but not automatic in the sense of Thurston. But for monoids, there is such an example.

13 Automatic Structures in General

Automatic groups (in the sense of Hodgson, Khoussainov and Nerode) are only a special case of an automatic structure.

Definition 13.1: Automatic Structure [33, 34, 45]**.** *A structure* $(A, R_1, R_2, \ldots, R_h, f_1, f_2, \ldots, f_k)$ *is called automatic iff* A *is a regular subset of some domain* Σ^* *and all relations* $R_1, R_2, \ldots, R_h$ *and functions* $f_1, f_2, \ldots, f_k$ *are automatic. More general, also structures which are isomorphic to automatic structures are called automatic.*

Example 13.2. The automatic structure $(0^*, Succ)$ with $Succ(x) = x0$ is isomorphic to the automatic structure of the natural numbers with successor $Succ(n) = n+1$. In this structure, the addition is not automatic, as the function $n \mapsto n + n$ has in that representation the graph $\{conv(0^n, 0^{2n}) : n \in \mathbb{N}\}$ which is not regular.

A Thurston automatic semigroup can be represented as an automatic structure consisting of a regular set R plus functions f_a mapping x to $x \circ a$ for every generator a of the semigroup. So $(\{0,1\}^*, x \mapsto 0x, x \mapsto 1x, \varepsilon)$ is an automatic structure representing the monoid of binary strings with concatenation; here is, however, instead of the full concatenation only the concatenation with the fixed strings 0 and 1 given.

$(\{0,1\}^*, \{0\}^*, <_{ll})$ is the set of binary strings with length-lexicographical order plus an additional set representing all those strings which only consist of 0s. Now $|x| < |y|$ is definable in this structure as

$$|x| < |y| \Leftrightarrow \exists z \in \{0\}^* \, [x <_{ll} z \wedge (z = y \vee z <_{ll} y)].$$

So a string in $\{0\}^*$ is the shortest string among the strings of its length; if one could compare the length of strings, then one could define

$$x \in \{0\}^* \Leftrightarrow \forall y <_{ll} x \, [|y| < |x|].$$

So comparison of sizes and $\{0\}^*$ are definable from each other using the ordering $<_{ll}$.

Description 13.3: Semirings and Rings. A commutative ring $(A, \oplus, \otimes, 0, 1)$ with 1 satisfies the following axioms:

- $\forall x, y \in A \, [x \oplus y = y \oplus x]$;
- $\forall x, y, z \in A \, [x \oplus (y \oplus z) = (x \oplus y) \oplus z]$;
- $\forall x, y \in A \, [x \otimes y = y \oplus x]$;
- $\forall x, y, z \in A \, [x \otimes (y \otimes z) = (x \otimes y) \otimes z]$;

- $\forall x, y, z \in A\,[x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)]$;
- $\forall x \in A\,[x \oplus 0 = x \wedge x \otimes 1 = x]$;
- $\forall x \in A \exists y \in A\,[x \oplus y = 0]$.

The laws listed here are the commutative and associative laws for $\oplus$ and $\otimes$, the law of distributivity, the law on the neutral elements and the existence of an inverse with respect to $\oplus$.

If one replaces the last axiom (existence of an inverse for addition) by $x \otimes 0 = 0 \otimes x = 0$ for all x, the structure $(A, \oplus, \otimes, 0, 1)$ is called a commutative semiring with 1. Note that the statement that $x \otimes 0 = 0$ is true in all rings, for semirings one needs to postulate it explicitly.

Furthermore, "commutative" refers to the multiplication $\otimes$. Also for rings which are not commutative, one assumes that the addition $\oplus$ is commutative. A structure without any law of commutativity is called a near-ring or a near-semiring which is defined as follows: Here a near-semiring $(A, \oplus, \otimes, 0)$ satisfies the associative laws for $\oplus$ and $\otimes$, the neutrality of 0 for the addition $\oplus$, one of the distributive laws $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$ or $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and $0 \otimes x = 0$ for all $x, y, z \in A$. A near-ring has the additional property that $(A, \oplus, 0)$ is a group, that is, that every element has an inverse for $\oplus$. Which of the distributive laws is used in a near-ring is a convention; each near-ring satisfying one distributive law can be transformed into a near-ring satisfying the other distributive law (by inverting the order of the operations in the near-ring).

An example for a ring is $(\mathbb{Z}, +, *, 0, 1)$; furthermore, for every $n \in \{2, 3, \ldots\}$, the structure $(\{0, 1, \ldots, n-1\}, +, *, 0, 1)$ with addition and multiplication taken modulo n is a ring. $(\mathbb{Z} \times \mathbb{Z}, +, *, (0,0), (1,1))$ with $+$ and $*$ carried out componentwise is a ring.

Example 13.4. Let $(F, +, \cdot)$ be the finite ring of addition and multiplication modulo a fixed number $r \in \{2, 3, \ldots\}$; so the domain F is $\{0, 1, \ldots, r-1\}$ and for $i, j \in F$ one defines that $i \oplus j = i + j$ if $i + j < r$ and $i \oplus j = i + j - r$ if $i + j \geq r$. Furthermore, $i \otimes j$ is the last digit of $i * j$ when written in an r-adic number system.

Now consider in the set $F^{\mathbb{N}}$ all functions $f : \mathbb{N} \to F$ which are eventually constant. Each such function can be represented by a string $a_0 a_1 \ldots a_n \in F^*$ where $f(m) = a_m$ for $m \leq n$ and $f(m) = a_n$ for $m > n$. Furthermore, let $rep(a_0 a_1 \ldots a_n) = a_0 a_1 \ldots a_m$ for the least $m \in \{0, 1, \ldots, n\}$ such that $a_n = a_k$ for all $k \in \{m, m+1, \ldots, n\}$. Let $A = \{rep(a_0 a_1 \ldots a_n) : n \in \mathbb{N} \wedge a_0, a_1, \ldots, a_n \in F\}$. Now one can define the pointwise operations $\oplus$ and $\otimes$ on A as follows:

Given $a_0 a_1 \ldots a_n$ and $b_0 b_1 \ldots b_m$, one says that $c_0 c_1 \ldots c_k = a_0 a_1 \ldots a_n \oplus b_0 b_1 \ldots b_m$

iff for all h, $c_{\min\{k,h\}} = a_{\min\{n,h\}} \oplus b_{\min\{m,h\}}$. Similarly for $\otimes$. These operations correspond to the pointwise addition and multiplication on eventually constant functions in $F^{\mathbb{N}}$. The element 0 represents the neutral element for the addition, that is, the function which is everywhere 0; the element 1 represents the neutral element for the multiplication, that is, the function which is everywhere 1.

The resulting structure is automatic. Furthermore, it is an example of a well-known type of mathematical structures, namely of an infinite ring.

Description 13.5: Orderings. An ordering $\sqsubset$ on a base set A is a relation which satisfies the following two axioms:

- $\forall x, y, z \in A\, [x \sqsubset y \wedge y \sqsubset z \Rightarrow x \sqsubset z]$;
- $\forall x [\neg x \sqsubset x]$.

The first law is called transitivity, the second irreflexivity. An ordering is called linear iff any two $x, y \in A$ are comparable, that is, satisfy $x \sqsubset y$ or $x = y$ or $y \sqsubset x$. Often one writes $x \sqsubseteq y$ for $x \sqsubset y \vee x = y$.

There are well-known automatic orderings, in particular $<_{lex}$ and $<_{ll}$ can be introduced on any set of strings. Also the ordering which says that $x \sqsubset y$ iff x is shorter than y can be introduced on every regular set. Another one $x \preceq y$ says that y extends x, that is, $y = xz$ for some z.

Exercise 13.6. *Consider* $(\{0\} \cdot \{0,1\}^* \cup \{1\}, \max_{lex}, \min_{lex}, 0, 1)$. *Show that this structure is an automatic semiring and verify the corresponding properties as well as the automaticity.*

Does this work for the maximum and minimum of any automatic linear ordering $\sqsubset$ *when the least element* 0 *and greatest element* 1 *exist?*

Given the commutative automatic semiring $(R, +, *, 0, 1)$, *consider the extension on* $R \times R \times R$ *with the componentwise operation* $+$ *and the new multiplication* $\odot$ *given by* $(x, y, z) \odot (x', y', z') = (x * x', y * y', x * z' + z * y')$? *Is this a semiring? Is* $\odot$ *commutative? What are the neutral elements for* $+$ *and* $\odot$ *in this ring? Prove the answer.*

A field is a ring with 1 where there exists for every $x \in A - \{0\}$ an $y \in A$ with $x \otimes y = 1$. The next result shows that infinite automatic fields do not exist.

Theorem 13.7. *There is no infinite automatic field.*

Proof. Assume that $(A, +, *, 0, 1)$ is an infinite automatic field; one can enrich this structure with the length-lexicographic ordering which is also automatic.

Now let $f(x)$ be the length-lexicographically first $y \in A$ such that for all a, a', b, b'

$\leq_{ll} x$ with $a \neq a'$ it holds that $(a - a') * y \neq b' - b$. The y exists. To see this, note that for each quadruple (a, a', b, b') with $a - a' \neq 0$ there is at most one y with $(a - a') * y = b' - b$. If there would be a second y' with the same property then $(a - a') * (y - y') = 0$ and now one can derive a contradiction. Let z be the multiplicative inverse of $y - y'$, that is, $(y - y') * z = 1$. Now $((a - a') * (y - y')) * z = 0$ and $(a - a') * ((y - y') * z) = a - a'$. However, by assumption, $a - a' \neq 0$, hence the law of associativity would be violated, a contradiction. Hence each quadruple (a, a', b, b') with $a \neq a'$ disqualifies only one y and as A has infinitely many y, there is some y which can be $f(x)$.

The function f is first-order defined using automatic parameters and therefore automatic.

Now consider $g(x, a, b) = a * f(x) + b$ and $h(x) = \max_{ll}\{g(x, a, b) : a, b \in A \wedge a, b \leq_{ll} x\}$. Both functions are automatic. Furthermore, if there are m elements in A up to x then there are at least m^2 elements up to $h(x)$; the reason is that if $(a, b) \neq (a', b')$ and $a, b \leq_{ll} x$ then one of the following two cases holds: in the case $a = a'$, it holds that $a * f(x) = a' * f(x)$ and $b \neq b'$ and this gives $a * f(x) + b \neq a' * f(x) + b'$; in the case $a \neq a'$, it holds that $(a - a') * f(x) \neq b' - b$ and again $a * f(x) + b \neq a' * f(x) + b'$ by a simple transformation of the inequality. So A has at least m^2 many elements of the form $a * f(x) + b$ with $a, b \leq_{ll} x$ and $a * f(x) + b \leq_{ll} h(x)$.

Now let $k \in \mathbb{N}$ be a constant so large that the field-elements A has at least 2 elements shorter than k and that $|h(x)| \leq |x| + k$ for all $x \in A$. Now let $A_r = \{x \in A : |x| \leq r \cdot k\}$. By induction, A_0 has at least 2^{2^0} elements and A_{r+1} has at least $2^{2^{r+1}}$ elements, as the number of elements in A_{r+1} is at least the square of the number of elements in A_r, hence $|A_{r+1}| \geq |A_r| \cdot |A_r| \geq 2^{2^r} \cdot 2^{2^r} = 2^{2^r + 2^r} = 2^{2^{r+1}}$. This would mean that the function $r \mapsto |A_r|$ grows at least double-exponentially in contradiction to the fact that all strings in A_r have length up to $r \cdot k$ so that there are at most $(1 + |\Sigma|)^{r \cdot k + 1}$ elements in A_r where Σ is the alphabet used. This contradiction shows that a field cannot be automatic. ∎

Description 13.8: Ordinals. In particular of interest are sets of ordinals. Small ordinals — and only they are interesting — can be viewed as expressions using finitely often exponentiation, power and addition and as constants ω and 1, where 1 stands for ω^0. Each exponentiation is of the form ω^α where α is an ordinal defined in the same way. If $\alpha \leq \beta$ then $\omega^\alpha \leq \omega^\beta$. Furthermore, $\alpha < \omega^\alpha$ for all the α considered here. Sums are always written with ω-powers in descending order. For example, $\omega^{\omega+\omega} + \omega^{\omega+1+1} + \omega^{1+1+1+1} + \omega^{1+1+1} + \omega^{1+1+1} + 1 + 1 + 1 + 1$. To simplify notation, repeated additions can be replaced by the corresponding natural numbers and $\omega^\alpha \cdot 5$ abbreviates that one has a sum of 5 identical terms ω^α. Above example gives $\omega^{\omega \cdot 2} + \omega^{\omega+2} + \omega^4 + \omega^3 \cdot 2 + 4$. One can compare two ordinals given by sums by looking at the first ω-power from above which has a different coefficient in both ordinals and then the ordinal with the larger coefficient is the larger one: $\omega^{\omega+2} + \omega^{\omega} \cdot 3 + \omega^{256} \cdot 373 < \omega^{\omega+2} + \omega^{\omega} \cdot 4 < \omega^{\omega+2} + \omega^{\omega+1} < \omega^{\omega+2} + \omega^{\omega+1} + \omega^{256}$.

There is furthermore an addition of ordinals. Given the ordinals as descending sums of ω-powers, the rule is to write the ω-powers of the second operand behind those of the first and then to remove all powers ω^{α} for which there is a power ω^{β} behind with $\alpha < \beta$. The coefficients of the highest ω-power occurring in the second operand are added in the case that this ω-power occurs in both operands. Here an example: $\omega^8 + \omega^5 + \omega^2$ plus $\omega^5 + \omega^4$ gives $\omega^8 + \omega^5 + \omega^5 + \omega^4 = \omega^8 + \omega^5 \cdot 2 + \omega^4$.

Note that the addition of the ordinals is not commutative. On one hand $\omega + 1 \neq \omega$ and on the other hand $1 + \omega = \omega$ by the cancellation-rule. Furthermore, one can introduce a multiplication for ordinals and show that the resulting structure is a near-semiring. As $(\mathbb{N}, +, \cdot, 0, 1)$ is a substructure of the ordinals whenever they go up to ω or beyond and as that structure is not automatic, the multiplication of ordinals is uninteresting for automatic structures.

So one investigates the ordinals with respect to the automaticity of their ordering and their addition. In particular, for given ordinal α one is interested in the linearly ordered set $\{\beta : \beta < \alpha\}$ and the following question had been investigated for years: For which α is the structure $(\{\beta : \beta < \alpha\}, <)$ isomorphic to an automatic linear order?

Example 13.9: Ordinals [19]**.** The ordinals below ω^4 are automatic. For this, recall that there is an automatic copy of $(\mathbb{N}, +, <)$. One can now represent each such ordinal by $conv(a_0, a_1, a_2, a_3)$ standing for $\omega^3 \cdot a_3 + \omega^2 \cdot a_2 + \omega \cdot a_1 + a_0$. Now the addition follows the following equation:

$$\begin{array}{l} conv(a_0, a_1, a_2, a_3) + \\ conv(b_0, b_1, b_2, b_3) = \end{array} \begin{cases} conv(a_0 + b_0, a_1, a_2, a_3) & \text{if } b_1 = 0,\ b_2 = 0,\ b_3 = 0; \\ conv(b_0, a_1 + b_1, a_2, a_3) & \text{if } b_1 > 0,\ b_2 = 0,\ b_3 = 0; \\ conv(b_0, b_1, a_2 + b_2, a_3) & \text{if } b_2 > 0,\ b_3 = 0; \\ conv(b_0, b_1, b_2, a_3 + b_3) & \text{if } b_3 > 0. \end{cases}$$

This function is automatic, as one can decide with automatic predicates which case applies and then form a convolution of functions which either copy one of the input-components or add two of them. Furthermore, the relation $<$ is first-order definable from the addition:

$$\begin{array}{l} conv(a_0, a_1, a_2, a_3) < conv(b_0, b_1, b_2, b_3) \Leftrightarrow \\ \quad \exists conv(c_0, c_1, c_2, c_3) \neq conv(0, 0, 0, 0) \\ \quad\quad [conv(a_0, a_1, a_2, a_3) + conv(c_0, c_1, c_2, c_3) = conv(b_0, b_1, b_2, b_3)]. \end{array}$$

Hence the ordinals below the power ω^4 form an automatic monoid with ordering. Delhommé [19] showed the following more general result.

Theorem 13.10: Delhommé's Characterisation of Automatic Ordinals [19]. *The following is equivalent for any ordinal α:*
(a) $\alpha < \omega^\omega$;
(b) $(\{\beta : \beta < \alpha\}, <)$ *is an automatic structure;*
(c) $(\{\beta : \beta < \alpha\}, +)$ *is an automatic structure.*
Here, in the case of (c), the domain of $+$ is the set of all pairs (β, γ) with $\beta + \gamma < \alpha$; in the case of α being an ω-power, this domain is the set $\{\beta : \beta < \alpha\} \times \{\gamma : \gamma < \alpha\}$.

Proof. Above it was shown for $\alpha = \omega^4$ that the ordinals below α with addition and ordering can be represented as an automatic structure. This proof generalises to all ω^n. Furthermore, if $\alpha \leq \omega^n$ for some $n \in \mathbb{N}$, then the set $\{\beta : \beta < \alpha\}$ can be defined in the automatic structure $(\{\beta : \beta < \omega^n\}, +, <)$ using α as a parameter and the resulting structure can serve to satisfy **(b)** and **(c)**, simultaneously. So assume by way of contradiction that $\alpha \geq \omega^\omega$ and that the structure $(\{\beta : \beta < \alpha\}, <)$ is automatic with a regular domain A and an automatic relation $<$. Add the string extension relation $\preceq$ on the domain to this structure as well as a relation to compare the length of strings. Assume that u_n represents ω^n in this structure. Now consider for all strings v with $|v| = |u_n|$ the sets $B_{u_n,v} = \{w \in A : v \preceq w \wedge w < u_n\}$.

Each set $B_{u_n,v}$ is defined in an uniform way and there is a finite automaton checking whether $w \in B_{u_n,v}$ when reading $conv(u_n, v, w)$. Furthermore, for $w_1, w_2 \in B_{u_n,v}$, there is a further automaton checking whether $w_1 < w_2$ in a way uniform in $conv(u_n, v, w_1, w_2)$. It is now easy to see that the structure $B'_{u_n,v} = (\{w' : vw' \in B_{u_n,v}\}, <')$ with $w'_1 <' w'_2 \Leftrightarrow vw_1 < vw_2$ only depends on the states of the automata recognising $B_{u_n,v}$ and $<$ after having read the first $|u_n|$ symbols of $conv(u_n, v, vw')$ and $conv(u_n, v, vw'_1, vw'_2)$, respectively. Hence there are only finitely many different such structures.

However, the set $\{w : w < u_n\}$ is for each u_n the union of finitely many sets $B_{u_n,v}$ and a finite set (of ordinals represented by strings shorter than u_n). One of the partially ordered sets $(B_{u_n,v}, <)$ must be isomorphic to $(\{\beta : \beta < \omega^n\}, <)$ and therefore the same holds for $(B'_{u_n,v}, <')$. As there are infinitely many such sets (different ordinals give non-isomorphic linearly ordered sets), this is a contradiction to the assumption that the structure of the ordinals below α is automatic. ∎

Exercise 13.11. *The above proof used one fact implicit: If $\{\beta : \beta < \omega^n\}$ is the union of finitely many sets $A_1, A_2, \ldots, A_m$ then one of the sets satisfies that $(A_k, <)$ is isomorphic to $(\{\beta : \beta < \omega^n\}, <)$. For $n = 1$ this is easily to be seen: Every infinite subset of the ordinals below ω is isomorphic to $(\mathbb{N}, <)$ and hence isomorphic to the set of ordinals below ω. For $n = 2$, this can be seen as follows: For each set A_k let*

$$\tilde{A}_k = \{i : \exists^\infty j\, [\omega \cdot i + j \in A_k]\}.$$

Each $i \in \mathbb{N}$ must appear in at least one set $\tilde{A}_k$. Hence there is a k for which $\tilde{A}_k$ is infinite. Now do the following: First, complete the case $n = 2$ by showing that for each k with $\tilde{A}_k$ being infinite the linearly ordered set $(A_k, <)$ is isomorphic to the set of ordinals below ω^2; second, generalise the whole proof to all $n \in \{3, 4, 5, 6, \ldots\}$.

Remarks 13.12. There are quite many structures for which one was able to show that they are not automatic: the multiplicative monoid of the natural numbers, of the rationals, of the positive rationals and so on. The polynomial ring in one or more variables over a finite field.

Furthermore, it had been investigated when a graph can be automatic, that is, be isomorphic to an automatic graph. Here (V, E) is an automatic graph iff V is a regular set and the set of edges E on V is an automatic relation. A graph on an infinite and countable set V is called a random graph iff the graph is undirected (that is, $(x, y) \in E \Leftrightarrow (y, x) \in E$ for all $x, y \in V$) and for any disjoint finite sets A, B there is a node x with $(x, y) \in E$ for all $y \in A$ and $(x, y) \notin E$ for all $y \in B$. It is known that all random graphs are isomorphic, that is, if (V, E) and (V', E') are random graphs then there is a bijection $f : V \to V'$ with $\forall x, y \in V\, [(x, y) \in E \Leftrightarrow (f(x), f(y)) \in E']$. Delhommé showed that no random graph is automatic.

Exercise 13.13. *Let $(G, \circ)$ be an automatic group and F be a regular subset of G. Is the graph (G, E) with $E = \{(x, y) : \exists z \in F\, [x \circ z = y]\}$ automatic?*

To which extent can the result be transferred to Thurston automatic groups? Consider the special cases for F being finite and F being infinite. In which cases are there Thurston automatic groups where it depends on the choice of F whether the corresponding graph is automatic or not?

Exercise 13.14. *Consider the following structure: For $a = (a_0, a_1, \ldots, a_n) \in \mathbb{N}^{n+1}$, let*

$$f_a(x) = \sum_{m=0}^{n} a_m \cdot \binom{x}{m}$$

and let F be the set of all so defined f_a (where n is not fixed). For which of the following orderings $<_k$ is $(F, <_k)$ an automatic partially ordered set?

(1) $a <_1 b \Leftrightarrow f_a \neq f_b$ and $f_a(x) < f_b(x)$ for the first x where they differ;
(2) $a <_2 b \Leftrightarrow \exists^\infty x\, [f_a(x) < f_b(x)]$;
(3) $a <_3 b \Leftrightarrow \forall^\infty x\, [f_a(x) < f_b(x)]$.

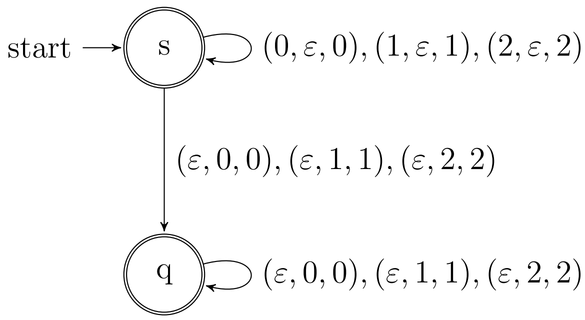
Give reasons for the answer.

14 Transducers and Rational Relations

There are two ways to generalise regular sets to regular relations: One is the notion of an automatic relation where all inputs are processed at the same speed (and exhausted shorter inputs padded with #) and the other notion is that of a rational relation which is defined below, where different inputs can be processed at different speed; rational relations are also called asynchronously automatic relations.

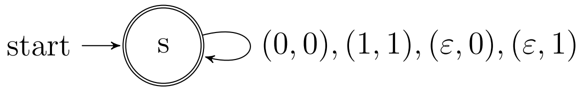
Definition 14.1: Rational Relation. *A relation $R \subseteq (\Sigma^*)^n$ is given by an non-deterministic finite state machine which can process n inputs in parallel and does not need to read them in the same speed. Transitions from one state p to a state q are labelled with an n-tuple $(w_1, w_2, \ldots, w_n)$ of words $w_1, w_2, \ldots, w_n \in \Sigma^*$ and the automaton can go along this transition iff for each input k the next $|w_k|$ symbols in the input are exactly those in the string w_k (this condition is void if $w_k = \varepsilon$) and in the case that the automaton goes on this transition, $|w_k|$ symbols are read from the k-th input word. A word $(x_1, x_2, \ldots, x_n)$ is in R iff there is a run of the machine which ends up in an accepting state after having reached the end-positions of all n words.*

Example 14.2: String Concatenation. The concatenation of strings over Σ^* is a rational relation. The following machine is given for $\Sigma = \{0, 1, 2\}$ and works for other alphabets correspondingly.



In the following graphical notation of a run on three input-words, the state is written always at that position which separates the read and not yet read parts of the input-words; the triple of input-words is $(01, 210, 01210)$ and the run is $(s01, s210, s01210) \Rightarrow (0s1, s210, 0s1210) \Rightarrow (01s, s210, 01s210) \Rightarrow (01q, 2q10, 012q10) \Rightarrow (01q, 21q0, 0121q0) \Rightarrow (01q, 210q, 01210q)$.

Example 14.3: Subsequence-Relation. A string x is a subsequence of y iff it can be obtained by from y by deleting symbols at some positions. The following one-state automaton recognises this relation for the binary alphabet $\{0, 1\}$.

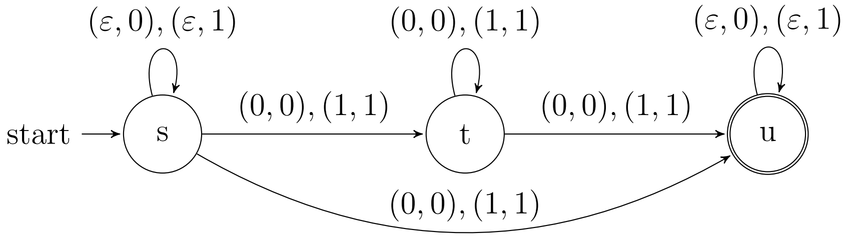


In general, there are one initial accepting state s with self-loops $s \to s$ labelled with (ε, a) and (a, a) for all $a \in \Sigma$.

So if $x = 0101$ and $y = 00110011$ then the automaton goes from s to itself with the transitions $(0,0),(\varepsilon,0),(1,1),(\varepsilon,1),(0,0),(\varepsilon,0),(1,1),(\varepsilon,1)$ and has afterwards exhausted both, x and y. As it is in an accepting state, it accepts this pair.

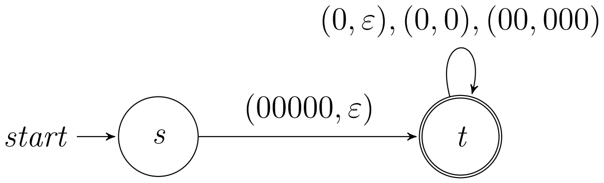
However, if $x = 00111$ and $y = 010101$ then the automaton cannot accept this pair: it gets stuck when processing it. After the first $(0,0)$, it has to use the transition $(\varepsilon, 1)$ in order to go on and can afterwards use the transition labelled $(0,0)$ again. But once this is done, the automaton has now on the x-side of the input 111 and on the y-side 101 so that it could go on with using $(1,1)$ once and would then have to use $(\varepsilon, 0)$ and afterwards $(1,1)$ again. However, now the automaton gets stuck with 1 being on the x-side while the y-side is exhausted. This is indeed also correct this way, as x is not a subsequence of y.

Example 14.4. This rational relation recognises that x is a non-empty substring of y, that is, $x \neq \varepsilon$ and $y = vxw$ for some $v, w \in \{0,1\}^*$. The automaton is the following.



When the automaton is in s or u, it parses the parts of x which are not in y while when going forward from s to u with perhaps cycling in t, the automaton compares x with the part of y which is equal to it in order to verify that x is a subword of y; furthermore, the automaton can do this only if x contains at least one symbol.

Exercise 14.5. *Rational relations got their name, as one can use them in order to express relations between the various inputs words which are rational. For example, one can look at the set of all* (x, y) *with* $|x| \geq \frac{2}{3}|y| + 5$. *This relation could be recognised by the following automaton (assuming that* $x, y \in \{0\}^*$*):*



Make automata which recognise the following relations:

(a) $\{(x, y) \in (0^*, 0^*) : 5 \cdot |x| = 8 \cdot |y|\}$*;*

(b) $\{(x, y, z) \in (0^*, 0^*, 0^*) : 2 \cdot |x| = |y| + |z|\}$*;*

(c) $\{(x, y, z) \in (0^*, 0^*, 0^*) : 3 \cdot |x| = |y| + |z| \vee |y| = |z|\}$*.*

Which automaton needs more than one state?

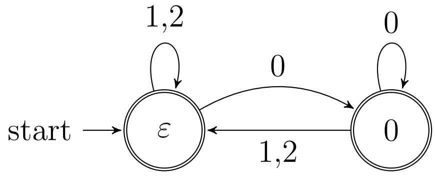
Description 14.6: Transducers. A rational function f mapping strings over Σ to strings over Σ is a function for which there is a rational relation R such that for each $x, y \in \Sigma^*$, $(x, y) \in R$ iff $x \in dom(f)$ and $f(x) = y$. Transducers are mechanisms to describe how to compute such a rational function and there are two types of them: Mealy machines and Moore machines. Both define the same class of functions.

A Mealy machine computing a rational function f is a non-deterministic finite automaton such that each transition is attributed with a pair (v, w) of strings and whenever the machine follows a transition $(p, (v, w), q)$ from state p to state q then one says that the Mealy machine processes the input part v and produces the output part w. If some run on an input x ends up in an accepting state and produces the output y, then every run on x ending up in an accepting state produces the same output and $f(x) = y$; if no run on an input x ends up in an accepting state then $f(x)$ is undefined.

Every automatic function is also a rational function and computed by a transducer, but not vice versa. For example, the function π preserving all symbols 0 and erasing the symbols $1, 2$ is given by the following one-state transducer: Starting state and accepting state is s, the transitions are $(s, (0, 0), s)$, $(s, (1, \varepsilon), s)$, $(s, (2, \varepsilon), s)$. This function π is not automatic.

Description 14.7: Moore machines [55]**.** Edward Moore [55] formalised functions computed by transducers by the concept of an automaton which is now known as a Moore machine. This is a non-deterministic finite automaton with possibly several starting states such that each state q owns a word w_q and each transition if of the form (q, a, p) for states q, p and elements $a \in \Sigma$. On input $a_1 a_2 \ldots a_n$, an accepting run is a sequence $(q_0, q_1, \ldots, q_n)$ of states starting with a starting state q_0 and ending in an accepting state q_n such that the transition-relation of the nfa permits for each $m < n$ to go from q_m to q_{m+1} on symbol a_{m+1} and the output produced by the run is the word $w_{q_0} w_{q_1} \ldots w_{q_n}$. A function f is computed by a Moore machine iff for each $x \in dom(f)$ there is an accepting run on input x with output $f(x)$ and for each string x and accepting run on input x with output y it holds that $f(x)$ is defined and $f(x) = y$.

First, consider the projection π from $\{0, 1, 2\}^*$ to $\{0\}^*$ which erases all $1, 2$ and preserves all 0; for example, $\pi(012012) = 00$. It needs a Moore machine having two states.



Second, let $f(a_1a_2\ldots a_n) = 012a_1a_1a_2a_2\ldots a_na_n012$. That is f doubles each symbol and places 012 before and after the output. The Moore machine given by the following table computes f:

state	starting	acc/rej	output	succ on 0	succ on 1	succ on 2
s	yes	rej	012	p, p'	q, q'	r, r'
p	no	rej	00	p, p'	q, q'	r, r'
q	no	rej	11	p, p'	q, q'	r, r'
r	no	rej	22	p, p'	q, q'	r, r'
s'	yes	acc	012012	–	–	–
p'	no	acc	00012	–	–	–
q'	no	acc	11012	–	–	–
r'	no	acc	22012	–	–	–

Now $f(0212)$ has the accepting run $sprqr'$ and this accepting run produces the output $w_sw_pw_rw_qw_{r'} = 012001100012$. The non-determinism mainly stems from the fact that the automaton does not know when the last symbol is read; therefore, it has non-deterministically choose between the states and their primed versions: s versus s', p versus p', q versus q' and r versus r'.

For an example with a more severe amount of non-determinism, consider the function g given by $g(a_1a_2\ldots a_n) = (\max(\{a_1, a_2, \ldots, a_n\}))^n$, so $g(\varepsilon) = \varepsilon$, $g(000) = 000$, $g(0110) = 1111$ and $g(00512) = 55555$. Now the Moore machine has to produce output in each state, but it has to choose in the first state which output to produce. So one has a starting state s with $w_s = \varepsilon$ and for each symbol a two states q_a and r_a with $w_{q_a} = w_{r_a} = a$. The states s and r_a are accepting, the states q_a are rejecting. The following transitions exist: (s, b, q_a) for all a, b with $b < a$, (s, a, q_a) for all a, (q_a, b, q_a) for all a, b with $b < a$ and (r_a, b, r_a) for all a, b with $b \leq a$. So when the Moore machine sees the first symbol and that is a 0, it has to decide which symbol a to write and there is no way to avoid this non-determinism.

Exercise 14.8. *Write down Mealy machines for the functions f and g from Description 14.7 of the Moore machines. For both, the alphabet can be assumed to be $\{0, 1, 2\}$.*

Exercise 14.9. *Determine the minimum number m such that every rational function can be computed by a non-deterministic Moore machine with at most m starting states. Give a proof that the number m determined is correct.*

Exercise 14.10. *Say that a Moore machine / Mealy machine is deterministic, if it has exactly one starting state and for it always reads one symbol from the input and for each state and each input symbol it has at most one transition which applies.*

Make a deterministic Moore machine and make also a deterministic Mealy machine which do the following with binary inputs: As long as the symbol 1 appears on the input, the symbol is replaced by 0; if at some time the symbol 0 appears, it is replaced by 1 and from then onwards all symbols are copied from the input to the output without a change.

So the function f computed by these machines satisfies $f(110) = 001$, $f(1111) = 0000$, $f(0) = 1$ and $f(110011) = 001011$.

Exercise 14.11. *Let the alphabet be $\{0, 1, 2\}$ and let $R = \{(x, y, z, u) : u$ has has $|x|$ many 0s, $|y|$ many 1s and $|z|$ many 2s$\}$. Is R a rational relation? Prove the result.*

Theorem 14.12 [57]. *Assume that $\Sigma_1, \Sigma_2, \ldots, \Sigma_m$ are disjoint alphabets. Furthermore, let π_k be the function which preserves all symbols from Σ_k and erases all other symbols. Then a relation $R \subseteq \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_m^*$ is rational iff there is a regular set P over a sufficiently large alphabet such that $(w_1, w_2, \ldots, w_n) \in R \Leftrightarrow \exists v \in P\, [\pi_1(v) = w_1 \wedge \pi_2(v) = w_2 \wedge \ldots \wedge \pi_m(v) = w_m]$.*

Proof. First, assume that a non-deterministic finite automaton recognises the rational relation R. Let Q be the set of states of this finite automaton and assume that Q is disjoint to all alphabets Σ_k. Furthermore, let the word $q_0 w_{1,1} w_{1,2} \ldots w_{1,m} q_1 w_{2,1} w_{2,2} \ldots w_{2,m} q_2 \ldots w_{n,1} w_{n,2} \ldots w_{n,m} q_n$ be in P iff q_0 is a starting state, q_n is an accepting state and for each $k < n$ the automaton goes from q_k on $(w_{k+1,1}, w_{k+1,2}, \ldots, w_{k+1,m})$ to q_{k+1}. In other words, P consists of representations of all accepting runs of the nfa on some input and if $v \in P$ then the input-tuple processed in this run is $(\pi_1(v), \pi_2(v), \ldots, \pi_m(v))$.

Second, for the converse direction, assume that a regular language P is given and that the dfa with starting state s and accepting states F is recognising P. Let Q be its states. Now one can make an nfa recognising the relation R by replacing every transition (p, a, q) of the original dfa with $(p, (\pi_1(a), \pi_2(a), \ldots, \pi_m(a)), q)$ in the new nfa. One has now to show that the new nfa recognises exactly the relation R.

Assume that there is a word $v = a_1 a_2 \ldots a_n \in P$ with $(w_1, w_2, \ldots, w_m) = (\pi_1(v), \pi_2(v), \ldots, \pi_m(v))$. There is a run $q_0 a_1 q_1 a_2 \ldots a_n q_n$ of the dfa which accepts the word v. Now one translates this run into $q_0\ (\pi_1(a_1), \pi_2(a_1), \ldots, \pi_m(a_1))\ q_1\ (\pi_1(a_2), \pi_2(a_2), \ldots, \pi_m(a_2))\ q_2 \ldots (\pi_1(a_n), \pi_2(a_n), \ldots, \pi_m(a_n))\ q_n$ and one can see that this is an accepting run of the nfa. Hence $(w_1, w_2, \ldots, w_n) \in R$.

Assume that $(w_1, w_2, \ldots, w_n) \in R$. Then the nfa accepts $(w_1, w_2, \ldots, w_n)$. This acceptance is witnessed by a run of the form q_0 $(\pi_1(a_1), \pi_2(a_1), \ldots, \pi_m(a_1))$ q_1 $(\pi_1(a_2), \pi_2(a_2), \ldots, \pi_m(a_2))$ q_2 $\ldots$ $(\pi_1(a_n), \pi_2(a_n), \ldots, \pi_m(a_n))$ q_n where q_0 is a starting state and q_n is an accepting state and the tuples between two states indicate the symbols read from the corresponding inputs. Then corresponds to an accepting run q_0 a_1 q_1 a_2 q_2 $\ldots$ a_n q_n on the original dfa which then accepts the word $v = a_1 a_2 \ldots a_n$. Hence, $v \in P$ and $(w_1, w_2, \ldots, w_m) = (\pi_1(v), \pi_2(v), \ldots, \pi_m(v))$. ∎

Description 14.13: Rational Structures. One could replace the requirement that relations are automatic by the requirement that relations are rational in oder to obtain a notion of rational structures. These are more general than automatic structures, but here various properties of automatic structures are lost:

- There are relations and functions which are first-order definable from rational relations without being a rational relation;
- There is no algorithm to decide whether a given first-order formula on automatic relations is true.

So the counterpart of the Theorem of Khoussainov and Nerode on automatic structures does not exist. While some properties are lost, the expressibility is in general higher. So various structures which are not automatic can be represented using rational relations. One example is given above: The monoid given by concatenation of strings over the alphabet $\{0, 1\}$.

Exercise 14.14. *There is a rational representation of the random graph. Instead of coding (V, E) directly, one first codes a directed graph (V, F) with the following properties:*

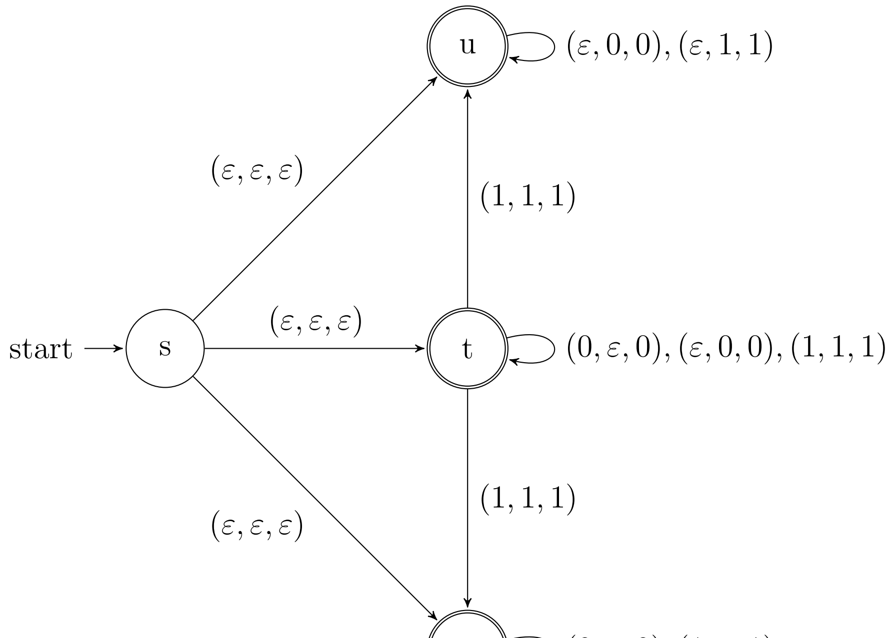
- *For each $x, y \in V$, if $(x, y) \in F$ then $|x| < |y|/2$;*
- *For each finite $W \subseteq V$ there is a y with $\forall x\,[(x, y) \in F \Leftrightarrow x \in W]$.*

This is done by letting $V = \{00, 01, 10, 11\}^+$ and defining that $(x, y) \in F$ iff there are n, m, k such that $y = a_0 b_0 a_1 b_1 \ldots a_n b_n$ and $a_m = a_k = 0$ and $a_h = 1$ for all h with $m < h < k$ and $x = b_m b_{m+1} \ldots b_{k-1}$. Give a transducer recognising F and show that this F satisfies the two properties above.

Now let $(x, y) \in E \Leftrightarrow (x, y) \in F \vee (y, x) \in F$. Show that (V, E) is the random graph by constructing to any given disjoint finite sets of strings A, B a string y longer than every string in A and B satisfying that for all $x \in A \cup B$, $(x, y) \in E$ iff $(x, y) \in F$ iff $x \in A$.

Example 14.15. The multiplicative monoid $(\mathbb{N} - \{0\}, *, 1)$ has a rational representation. Note that every natural number is given by its primefactors: So $(n_1, n_2, \ldots, n_k)$ with $n_k > 0$ represents the number $2^{n_1} * 3^{n_2} * \ldots * {p_k}^{n_k}$ and the empty vector represents 1. So 36 is represented by $(2, 2)$ (for $2^2 * 3^2$) and 3 is represented by $(0, 1)$. Now one has that $36 * 3$ is represented by $(2, 3)$ which is the componentwise sum of $(2, 2)$ and $(0, 1)$. Furthermore, 30 is represented by $(1, 1, 1)$ so that $36 * 30$ needs that one adjust the length of the shorter vector before one does the componentwise addition: $36 * 30$ is represented by $(2, 2) + (1, 1, 1) = (2, 2, 0) + (1, 1, 1) = (3, 3, 1)$. In other words the set $\mathbb{N} - \{0\}$ with multiplication and the finite vectors of natural numbers with a non-zero last component with the operation of componentwise addition (where 0 is invoked for missing components) are isomorphic monoids.

In the next step, one has to represent each vector $(n_1, n_2, \ldots, n_k)$ by a string, so one takes $0^{n_1} 1 0^{n_2} 1 \ldots 0^{n_k} 1$ and represents the empty vector (standing for the natural number 1) by ε. Now $36 * 3 = 108$ is represented by $001001 * 101 = 0010001$ and $36 * 30 = 1080$ is represented by $001001 * 010101 = 0001000101$. The domain is $\{\varepsilon\} \cup \{0, 1\}^* \cdot \{01\}$ and is therefore regular. The following automaton recognises the graph of the rational relation $*$:



When verifying that $x * y = z$, t is the node which is used as long as x and y are both not exhausted; u is the node to be used when the end of x is reached while the end of y has still to be found while v is the node to be used when the end of y has been reached while the end of x has still to be found. There are transitions on empty tuples from s to all of these three nodes as the two extreme cases that one of x and y is ε need to be covered as well.

Selftest 14.16. *Let f be an automatic function from a regular set A to a regular set B. Let $B_n = \{y \in B : y = f(x)$ for exactly n words $x \in A\}$ and let $B_\infty = \{y \in B : y = f(x)$ for infinitely many $x \in A\}$. Which of the sets $B_0, B_1, \ldots, B_\infty$ are regular?*

Selftest 14.17. *Assume that a group is generated by elements a, b, c and their inverses $\overline{a}, \overline{b}, \overline{c}$ and has the following rules: $a \circ b = c \circ a$, $a \circ c = b \circ a$, $b \circ c = c \circ b$ and correspondingly for the inverses, that is, $a \circ \overline{b} = \overline{c} \circ a$, $\overline{a} \circ b = c \circ \overline{a}$ and so on.*

Show that this group is automatic by providing an automatic representation and explain why the group operation $conv(x, y) \mapsto x \circ y$ is an automatic function (with two inputs) in this representation.

Selftest 14.18. *Let $L = \{00, 11\}^*$ be a regular language over the alphabet $\{0, 1, 2, 3\}$. Determine the syntactic monoid G_L for this language.*

Selftest 14.19. *Let $G = \{a^* \cup \overline{a}^*\} \cdot \{b^* \cup \overline{b}^*\}$ be a representation for the Thurston automatic group generated by a, b and the inverses $\overline{a}, \overline{b}$ with the rule $a \circ b = b \circ a$. Let L be the set of all strings over $\{a, \overline{a}, b, \overline{b}\}$ which are equivalent to the neutral element ε in this group.*

What is the complexity of L? (a) regular, (b) context-free and not regular, (c) context-sensitive and not context-free, (d) recursively enumerable and not context-sensitive.

Give a justification for the taken choice.

Selftest 14.20. *Let $\{L_d : d \in I\}$ and $\{H_e : e \in J\}$ be two automatic families with regular sets of indices I, J. Prove or disprove the following claim.*

Claim. There are an automatic relation $R \subseteq I \times I$ and an automatic function $f : R \to J$ with domain R such that for all $d, d' \in I$ the following statement holds: if there is an $e \in J$ with $L_d \cap L_{d'} = H_e$ then $(d, d') \in R$ and $H_{f(d,d')} = H_e$ else $(d, d') \notin R$.

Selftest 14.21. *Is every function which is computed by a non-deterministic Moore machine also computed by a deterministic Moore machine?*

If the answer is "yes" then explain how the Moore machine is made deterministic; if the answer is "no" then give an example of a function which is computed only by a non-deterministic Moore machine and not by a deterministic one.

Selftest 14.22. *Construct a Mealy machine which recognises the following function: $f(x)$ doubles every 0 and triples every 1 if x does not contain a 2; $f(x)$ omits all 0 and 1 from the input if x contains a 2.*

Sample outputs of f are $f(01) = 00111$, $f(01001) = 001110000111$, $f(021) = 2$ and $f(012210012) = 222$.

Solution for Selftest 14.16. All of the sets $B_0, B_1, \ldots, B_\infty$ are regular. The reason is that one can first-order define each of the sets using automatic functions (like f) and relations (like membership in A and length-lexicographic order). For example,

$$y \in B_0 \Leftrightarrow y \in B \wedge \forall x \in A\, [y \neq f(x)]$$

and

$$\begin{aligned} y \in B_2 \quad \Leftrightarrow \quad & \exists x_1, x_2 \in A\, \forall x \in A \\ & [f(x_1) = y \wedge f(x_2) = y \wedge x_1 \neq x_2 \wedge f(x) = y \rightarrow (y = x_1 \vee y = x_2)]. \end{aligned}$$

The formula for B_∞ has to be a bit different, as one has to say that there are infinitely many x which are mapped to y. For this one assumes that A is infinite, as otherwise $B_\infty = \emptyset$. Now the formula is

$$y \in B_\infty \Leftrightarrow \forall x \in A\, \exists x' \in A\, [x <_{ll} x' \wedge f(x') = y].$$

It is okay to introduce new automatic parameters (like the length-lexicographic ordering on A) in order to show that some set or relation or function is regular / automatic by providing the corresponding first-order definition.

Solution for Selftest 14.17. Take any automatic representation $(A, +)$ of the integers and note that the set $B = \{x : \exists y\, [y+y = x]\}$ of the even integers is regular. Now represent the group $(G, \circ)$ by $\{conv(i,j,k) : i,j,k \in A\}$ where $conv(i,j,k)$ stands for $a^i \circ b^j \circ c^k$. Now $conv(i,j,k) \circ conv(i',j',k')$ is $conv(i+i', j+j', k+k')$ in the case that i' is even and $conv(i+i', k+j', j+k')$ in the case that i' is odd. As B is regular, this case-distinction is automatic; furthermore, the addition is automatic in A and can therefore be carried out on the components.

Solution for Selftest 14.18. For the language $\{00, 11\}^*$, one has first to make the minimal dfa which has four states, namely s, z, o, t. Its transition table is the following (where s is the starting state):

state	acc/rej	0	1	2	3
s	accept	z	o	t	t
z	reject	s	t	t	t
o	reject	t	s	t	t
t	reject	t	t	t	t

The corresponding monoid has the following function f_u where for each f_u only one representative word u is taken.

word u	$f_u(s)$	$f_u(z)$	$f_u(o)$	$f_u(t)$
ε	s	z	o	t
0	z	s	t	t
00	s	z	t	t
001	o	t	t	t
0011	s	t	t	t
01	t	o	t	t
011	t	s	t	t
0110	t	z	t	t
1	o	t	s	t
10	t	t	z	t
100	t	t	s	t
1001	t	t	o	t
11	s	t	o	t
110	z	t	t	t
2	t	t	t	t

Solution for Selftest 14.19. The answer should be (c) "context-sensitive and not context-free". Let L_a be the set of all words in $\{a, \overline{a}, b, \overline{b}\}^*$ which have as many a as $\overline{a}$ and L_b be the set of all words in $\{a, \overline{a}, b, \overline{b}\}^*$ which have as many b as $\overline{b}$. Then $L = L_a \cap L_b$, thus L is the intersection of two context-free languages and therefore context-sensitive or context-free or regular.

Note that all levels of the Chomsky hierarchy are closed with respect to intersection with regular sets. Now one forms the set $L \cap a^*(\overline{b}\overline{a})^*b^*$. This set consists of all words of the form $a^n(\overline{b}\overline{a})^n b^n$ and this is a well-known example of a context-sensitive language which is not context-free. Therefore the language L cannot be regular and cannot be context-free; so context-sensitive is the right level.

Solution for Selftest 14.20. The claim is true. The main idea is that the function f plus its domain is first-order definable from automatic parameters. Indeed, one can introduce a relation R' and then derive R, f from R' as follows:

$$\begin{aligned}(d, d', e) \in R' &\Leftrightarrow d, d' \in I \wedge e \in J \wedge \forall x\, [x \in H_e \Leftrightarrow x \in L_d \wedge x \in L_{d'}];\\ (d, d') \in R &\Leftrightarrow \exists e \in J\, [R'(d, d', e)];\\ f(d, d') = e &\Leftrightarrow R'(d, d', e) \wedge \forall e' \in J\, [R'(d, d', e') \Rightarrow e \leq_{ll} e'].\end{aligned}$$

Here again one uses the automatic length-lexicographic ordering and the automaticity of the membership problem of the corresponding automatic families.

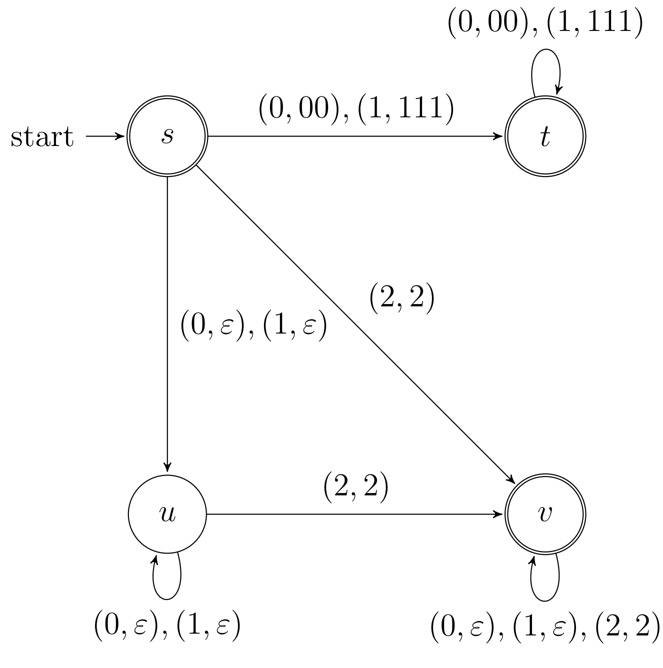
Solution for Selftest 14.21. The answer is "no" and the reason is that a non-deterministic Moore machine can anticipate some information which the deterministic

Moore machine cannot anticipate.

For example, a Moore machine should map any input $a_0 a_1 \ldots a_n$ to $(a_n)^{n+1}$, that is, replace all a_m by the last digit a_n. For this the Moore machine needs non-deterministically to anticipate what a_n is. So the Moore machine has a start state s without any output and for each symbol $a \in \Sigma$ it has two states r_a (rejecting) and q_a (accepting). Now on the first symbol b the Moore machine non-deterministically chooses a and if $b = a$ then it goes to q_a else it goes to r_a. On each further symbol c, if $a = c$ then the machine goes to q_a else it goes to r_a. Both states r_a and q_a output in each cycle one symbol a. If the last input symbol is a then the automaton will be in the accepting state q_a else it will be in the rejecting state r_a. So if the input ends with a the run is accepting and the output is correct; if the input does not end with a then the run ends in a rejecting state and the output is not valid.

A deterministic Moore machine cannot compute this function. If the Moore machine sees an input 0, it needs to respond with a 0 immediately, as it otherwise would not map 0 to 0, hence it goes on 0 to a state with output 0. If then a 1 follows, the output has to be 11, what is impossible for the deterministic Moore machine to do, as it has already written a 0.

Solution for Selftest 14.22.



15 Models of Computation

Since the 1920ies and 1930ies, mathematicians investigated how to formalise the notion of computation in an abstract way. These notions are the Turing machine, the register machine and the μ-recursive functions.

Definition 15.1: Turing Machine [76]**.** A Turing machine is a model to formalise on how to compute an output from an input. The basic data storage is an infinite tape which has at one place the input word on the tape with infinitely many blancs before and after the word. The Turing machine works on this work in cycles and is controlled by states, similarly to a finite automaton. It also has a head position on the tape. Depending on the state on and the symbol under the head on the tape, the Turing machine writes a new symbol (which can be the same as before), chooses a new state and moves either one step left or one step right. One special state is the halting state which signals that the computation has terminated; in the case that one wants several outcomes to be distinguishable, one can also have several halting states, for example for "halt and accept" and "halt and reject". These transitions are noted down in a table which is the finite control of the Turing Machine; one can also see them as a transition function δ.

One says that the Turing machine computes a function f from Σ^* to Σ^* iff the head before the computation stands on the first symbol of the input word, then the computation is performed and at the end, when the machine goes into the halting state, the output is the content written on the Turing machine tape. In the case that for some input w the Turing machine never halts but runs forever then $f(w)$ is undefined. Thus Turing machines compute partial functions.

Note that Turing machines, during the computation, might use additional symbols, thus their tape alphabet Γ is a superset of the alphabet Σ used for input and output. Formally, a Turing machine is a tuple $(Q, \Gamma, \sqcup, \Sigma, \delta, s, F)$ where Q is the set of states with start state s and the set of halting states F; Σ is the input alphabet, Γ the tape alphabet and $\sqcup$ the special space symbol in $\Gamma - \Sigma$; so $\Sigma \subset \Gamma$. δ is the transition functions and maps pairs from $(Q - F) \times \Gamma$ to triples from $Q \times \Gamma \times \{left, right\}$. δ can be undefined on some combinations of inputs; if the machine runs into such a situation, the computation is aborted and its value is undefined.

Example 15.2. The following Turing machine maps a binary number to its successor, so 100 to 101 and 111 to 1000.

state	symbol	new state	new symbol	movement
s	0	s	0	right
s	1	s	1	right
s	$\sqcup$	t	$\sqcup$	left
t	1	t	0	left
t	0	u	1	left
t	$\sqcup$	u	1	left

This table specifies the function δ of the Turing machine $(\{s, t, u\}, \{0, 1, \sqcup\}, \sqcup, \{0, 1\}, \delta, s, \{u\})$. At the beginning, the head of the Turing machine stands on the first symbol of the input, say on the first 1 of 111. Then the Turing machine moves right until it reaches a blanc symbol $\sqcup$. On $\sqcup$ it transits into t and goes one step to the left back onto the input number. Then it transforms each 1 into a 0 and goes left until it reaches a 0 or $\sqcup$. Upon reaching this symbol, it is transformed into a 1 and the machine halts.

Exercise 15.3. *Construct a Turing machine which computes the function $x \mapsto 3x$ where the input x as well as the output are binary numbers.*

Exercise 15.4. *Construct a Turing machine which computes the function $x \mapsto x + 5$ where the input x as well as the output are binary numbers.*

In the numerical paradigm, one considers natural numbers as primitives. For this, one could, for example, identify the numbers with strings from $\{0\} \cup \{1\} \cdot \{0, 1\}^*$.

If one wants to use all binary strings and make a bijection to the natural numbers, one would map a string $a_1 a_2 \ldots a_n$ the value $b-1$ of the binary number $b = 1a_1 a_2 \ldots a_n$, so ε maps to 0, 0 maps to 1, 1 maps to 2, 00 maps to 3 and so on. The following table gives some possible identifications of members with $\mathbb{N}$ with various ways to represent them.

decimal	binary	bin words	ternary	ter words
0	0	ε	0	ε
1	1	0	1	0
2	10	1	2	1
3	11	00	10	2
4	100	01	11	00
5	101	10	12	01
6	110	11	20	02
7	111	000	21	10
8	1000	001	22	11
9	1001	010	100	12
10	1010	011	101	20

Now one defines a register machine as a machine working on numbers and not on strings. Here the formal definition.

Description 15.5: Register Machine. A register machine consists of a program and a storage consisting of finitely many registers $R_1, R_2, \ldots, R_n$. The program has line numbers and one can jump from one to the next; if no jump instruction is given, after an instruction, the next existing line number applies. The following types of instructions can be done:

- $R_i = c$ for a number c;
- $R_i = R_j + c$ for a number c;
- $R_i = R_j + R_k$;
- $R_i = R_j - c$ for a number c, where the number 0 is taken if the result would be negative;
- $R_i = R_j - R_k$, where the number 0 is taken if the result would be negative;
- If $R_i = c$ Then Goto Line k;
- If $R_i = R_j$ Then Goto Line k;
- If $R_i < R_j$ Then Goto Line k;
- Goto Line k;
- Return(R_i), finish the computation with content of Register R_i.

One could be more restrictive and only allow to add or subtract the constant 1 and to compare with 0; however, this makes the register programs almost unreadable. The register machine computes a mapping which maps the contents of the input registers to the output; for making clear which registers are input and which are not, one could make a function declaration at the beginning. In addition to these conventions, in the first line of the register program, one writes the name of the function and which registers are read in as the input. The other registers need to be initialised with some values by the program before they are used.

Example 15.6. The following program computes the product of two numbers.

Line 1: Function Mult(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;

Line 4: If $R_3 = R_1$ Then Goto Line 8;
Line 5: $R_4 = R_4 + R_2$;
Line 6: $R_3 = R_3 + 1$;
Line 7: Goto Line 4;
Line 8: Return(R_4).

The following program computes the remainder of two numbers.

Line 1: Function Remainder(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;
Line 4: $R_5 = R_4 + R_2$;
Line 5: If $R_1 < R_5$ Then Goto Line 8;
Line 6: $R_4 = R_5$;
Line 7: Goto Line 4;
Line 8: $R_3 = R_1 - R_4$;
Line 9: Return(R_3).

The program for integer division is very similar.

Line 1: Function Divide(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;
Line 4: $R_5 = R_4 + R_2$;
Line 5: If $R_1 < R_5$ Then Goto Line 9;
Line 6: $R_3 = R_1 + 1$;
Line 7: $R_4 = R_5$;
Line 8: Goto Line 4;
Line 9: Return(R_3).

Exercise 15.7. *Write a program P which computes for input x the value $y = 1+2+3+\ldots+x$.*

Exercise 15.8. *Write a program Q which computes for input x the value $y = P(1)+P(2)+P(3)+\ldots+P(x)$ for the program P from the previous exercise.*

Exercise 15.9. *Write a program O which computes for input x the factorial $y = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot x$. Here the factorial of 0 is 1.*

Description 15.10: Subprograms. Register machines come without a management for local variables. When writing subprograms, they behave more like macros:

One replaces the calling text with a code of what has to be executed at all places inside the program where the subprogram is called. Value passing into and the function and returning back is implemented; registers inside the called function are renumbered to avoid clashes. Here the example of the function "Power" using the function "Mult".

Line 1: Function Power(R_5, R_6);
Line 2: $R_7 = 0$;
Line 3: $R_8 = 1$;
Line 4: If $R_6 = R_7$ Then Goto Line 8;
Line 5: $R_8 = \text{Mult}(R_8, R_5)$;
Line 6: $R_7 = R_7 + 1$;
Line 7: Goto Line 4;
Line 8: Return(R_8).

Putting this together with the multiplication program only needs some code adjustments, the registers are already disjoint.

Line 1: Function Power(R_5, R_6);
Line 2: $R_7 = 0$;
Line 3: $R_8 = 1$;
Line 4: If $R_6 = R_7$ Then Goto Line 16;
Line 5: $R_1 = R_5$; // Initialising the Variables used
Line 6: $R_2 = R_8$; // in the subfunction
Line 7: $R_3 = 0$; // Subfunction starts
Line 8: $R_4 = 0$;
Line 9: If $R_3 = R_1$ Then Goto Line 13;
Line 10: $R_4 = R_4 + R_2$;
Line 11: $R_3 = R_3 + 1$;
Line 12: Goto Line 9;
Line 13: $R_8 = R_4$; // Passing value back, subfunction ends
Line 14: $R_7 = R_7 + 1$;
Line 15: Goto Line 4;
Line 16: Return(R_8).

This example shows that it is possible to incorporate subfunctions of this type into the main function; however, this is more difficult to read and so the subfunctions are from now on preserved. Note that due to the non-implementation of the saving of the line number, the register machines need several copies of the called function in the case that it is called from different positions, for each position one. In short, subprograms are more implemented like macros than like functions in programming. Though this restriction is there, the concept is useful.

The next paragraph shows how to code a Turing machine using a one-sided tape (with a starting point which cannot be crossed) in a register machine.

Description 15.11: Coding and Simulating Turing Machines. If one would have $\Gamma = \{0, 1, 2, \ldots, 9\}$ with 0 being the blanc, then one could code a tape starting at position 0 as natural numbers. The leading zeroes are then all blanc symbols on the tape. So, in general, one represent the tape by numbers in a system with $|\Gamma|$ many digits which are represented by $0, 1, \ldots, |\Gamma| - 1$. The following functions in register programs show how to read out and to write a digit in the tape.

Line 1: Function Read(R_1, R_2, R_3); // $R_1 = |\Gamma|$, R_2 = Tape, R_3 = Position
Line 2: $R_4 = \text{Power}(R_1, R_3)$;
Line 3: $R_5 = \text{Divide}(R_2, R_4)$;
Line 4: $R_6 = \text{Remainder}(R_5, R_1)$;
Line 5: Return(R_6). // Return Symbol

The operation into the other direction is to write a digit onto the tape.

Line 1: Function Write(R_1, R_2, R_3, R_4); // $R_1 = |\Gamma|$, R_2 = Tape, R_3 = Position, R_4 = New Symbol
Line 2: $R_5 = \text{Power}(R_1, R_3)$;
Line 3: $R_6 = \text{Divide}(R_2, R_5)$;
Line 4: $R_7 = \text{Remainder}(R_6, R_1)$;
Line 5: $R_6 = R_6 + R_4 - R_7$;
Line 6: $R_8 = \text{Mult}(R_6, R_5)$;
Line 7: $R_9 = \text{Remainder}(R_2, R_5)$;
Line 8: $R_9 = R_9 + R_8$;
Line 9: Return(R_9). // Return New Tape

For the general implementation, the following assumptions are made:

- Input and Output is, though only using the alphabet Σ, already coded in the alphabet Γ which is a superset of Σ.
- When representing the symbols on the tape, 0 stands for $\sqcup$ and Σ is represented by $1, 2, \ldots, |\Sigma|$ and the other symbols of Γ are represented by the next numbers.
- The starting state is 0 and the halting state is 1 — it is sufficient to assume that there is only 1 for this purpose.

- The Turing machine to be simulated is given by four parameters: R_1 contains the size of Γ, R_2 contains the number $|Q|$ of states, R_3 contains the Turing Table which is an array of entries from $\Gamma \times Q \times \{left, right\}$ organised by indices of the form $q \cdot |\Gamma| + \gamma$ for state q and symbol γ (in numerical coding). The entry for $(\gamma, q, movement)$ is $\gamma \cdot |Q| \cdot 2 + q \cdot 2 + movement$ where $movement = 1$ for going right and $movement = 0$ for going left. This table is read out like the tape, but it cannot be modified by writing. R_4 contains the Turing tape which is read and updated.
- Input and Output are on tapes of the form $\sqcup w \sqcup^\infty$ and the Turing machine cannot go left on 0, it just stays where it is ($0 - 1 = 0$ in this coding). The register R_5 contains the current tape position.
- R_6 contains the current symbol and R_7 contains the current state and R_8 the current instruction.
- The register machine simulating the Turing machine just runs in one loop and if the input is a coding of the input word and the Turing machine runs correctly then the output is a coding of the tape at the output.

So the main program of the simulation is the following.

Line 1: Function Simulate(R_1, R_2, R_3, R_4);
Line 2: $R_5 = 1$;
Line 3: $R_7 = 0$;
Line 4: $R_9 = \text{Mult}(\text{Mult}(2, R_2), R_1)$; // Size of Entry in Turing table
Line 5: $R_6 = \text{Read}(R_1, R_4, R_5)$; // Read Symbol
Line 6: $R_8 = \text{Read}(R_9, R_3, \text{Mult}(R_1, R_7) + R_6)$; // Read Entry
Line 7: $R_{10} = \text{Divide}(R_8, \text{Mult}(R_2, 2))$; // Compute New Symbol
Line 8: $R_4 = \text{Write}(R_1, R_4, R_5, R_{10})$; // Write New Symbol
Line 9: $R_7 = \text{Remainder}(\text{Divide}(R_8, 2), R_2)$; // Compute New State
Line 10: If $R_7 = 1$ Then Goto Line 13; // If State is Halting, Stop
Line 11: $R_5 = R_5 + \text{Mult}(2, \text{Remainder}(R_8, 2)) - 1$; // Move Head
Line 12: Goto Line 5;
Line 13: Return(R_4).

This simulation shows that for fixed alphabet Σ, there is a universal Register machine which computes a partial function ψ such that $\psi(i, j, k, x)$ is the unique $y \in \Sigma^*$ for which the simulation of the Turing machine given by tape alphabet of size i, number of states j and table k maps the tape $\sqcup x \sqcup^\infty$ to the tape $\sqcup y \sqcup^\infty$ and halts. The three parameters i, j, k are usually coded into one parameter e which is called the Turing program.

Theorem 15.12. *Every Turing machine can be simulated by a register machine and there is even one single register machine which simulates for input* (e, x) *the Turing machine described by* e*; if this simulation ends with an output* y *in the desired form then the register machine produces this output; if the Turing machine runs forever, so does the simulating register machine.*

Exercise 15.13. *Explain how one has to change the simulation of the Turing machine in order to have a tape which is in both directions infinite.*

Alan Turing carried out the above simulations inside the Turing machine world. This permitted him to get the following result.

Theorem 15.14: Turing's Universal Turing Machine [76]**.** There is one single Turing machine which simulates on input (e, x) the actions of the e-th Turing machine with input x.

In the same way that one can simulate Turing machines by register machines, one can also simulate register machines by Turing machines. Modulo minor adjustments of domain and range (working with natural numbers versus working with words), the two concepts are the same.

Theorem 15.15. *If one translates domains and ranges in a canonical way, then the partial functions from* Σ^* *to* Σ^* *computed by a Turing machine are the same as the partial functions from* $\mathbb{N}$ *to* $\mathbb{N}$ *computed by a register machine.*

Another way to define functions is by recursion. The central notion is that of a primitive recursive function, which is also defined by structural induction.

Definition 15.16: Primitive Recursive Functions [71]**.** *First, the following base functions are primitive recursive.*

Constant Function: *The function producing the constant* 0 *without any inputs is primitive recursive.*

Successor Function: *The function* $x \mapsto x + 1$ *is primitive recursive.*

Projection Function: *Each function of the form* $x_1, \ldots, x_n \mapsto x_m$ *for some* m, n *with* $m \in \{1, \ldots, n\}$ *is primitive recursive.*

Second, there are two ways to define inductively new primitive recursive functions from others.

Composition: *If $f : \mathbb{N}^n \to \mathbb{N}$ and $g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}$ are primitive recursive, so is $x_1, \ldots, x_m \mapsto f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_n))$.*

Recursion: *If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are primitive recursive then there is also a primitive recursive function h with $h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ and $h(y + 1, x_1, \ldots, x_n) = g(y, h(y, x_1, \ldots, x_n), x_1, \ldots, x_n)$.*

In general, this says that one can define primitive recursive functions by some base cases, concatenation and recursion in one variable.

Example 15.17. The function $h(x) = pred(x) = x - 1$, with $0 - 1 = 0$, is primitive recursive. One defines $pred(0) = 0$ and $pred(y+1) = y$, more precisely $pred$ is defined using f, g with $f(x) = 0$ and $g(y, pred(x)) = y$.

Furthermore, the function $x, y \mapsto h(x, y) = x - y$ is primitive recursive, one defines $x - 0 = x$ and $x - (y+1) = pred(x - y)$. This definition uses implicit that one can instead of $h(x, y)$ use $\tilde{h}(y, x)$ which is obtained by swapping the variables; now $h(y, x) = \tilde{h}(second(y, x), first(y, x))$ where $first, second$ pick the first and second input variable of two inputs. By induction one has $\tilde{h}(0, x) = x$ and $\tilde{h}(y + 1, x) = pred(y, x)$, so $\tilde{h}(y, x) = x - y$.

Now one can define $equal(x, y) = 1 - (x - y) - (y - x)$ which is 1 if x, y are equal and which is 0 if one of the terms $x - y$ and $y - x$ is at least 1.

Another example is $x + y$ which can be defined inductively by $0 + y = y$ and $(x + 1) + y = succ(x + y)$, where $succ : z \mapsto z + 1$ is one of the base functions of the primitive recursive functions.

Exercise 15.18. *Prove that every function of the form $h(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n + b$ with fixed parameters $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$ is primitive recursive.*

Exercise 15.19. *Prove that the function $h(x) = 1 + 2 + \ldots + x$ is primitive recursive.*

Exercies 15.20. *Prove that the multiplication $h(x, y) = x \cdot y$ is primitive recursive.*

Primitive recursive functions are always total. Thus one can easily derive that there is no primitive recursive universal function for them. In the case that there would be a function $f(e, x)$ which is primitive recursive such that for all primitive recursive functions g with one input there is an e such that $\forall x\, [g(x) = f(e, x)]$ then one could easily make a primitive recursive function which grows faster than all of these:

$$h(x) = 1 + f(0, x) + f(1, x) + \ldots + f(x, x).$$

To see that this function is primitive recursive, one first considers the two place function

$$\tilde{h}(y, x) = 1 + f(0, x) + f(1, x) + \ldots + f(y, x)$$

by defining $\tilde{h}(0, x) = 1 + f(0, x)$ and $\tilde{h}(y + 1, x) = \tilde{h}(y, x) + f(y + 1, x)$. Bow $h(x) = \tilde{h}(x, x)$. Thus one has that there is no universal primitive recursive function for all primitive recursive functions with one input; however, it is easy to construct such a function computed by some register machine. Ackermann [1] was able to give a recursive function defined by recursion over several variables which is not primitive recursive. In the subsequent literature, several variants were studied; generally used is the following form of his function:

- $f(0, y) = y + 1$;
- $f(x + 1, 0) = f(x, 1)$;
- $f(x + 1, y + 1) = f(x, f(x + 1, y))$.

So it is a natural question on how to extend this notion. This extension is the notion of μ-recursive functions; they appear first in the Incompleteness Theorem of Gödel [30] where he characterised the recursive functions on the way to his result that one cannot make a complete axiom system for the natural numbers with $+$ and $\cdot$ which is enumerated by an algorithm.

Definition 15.21: Partial recursive functions [30]. *If $f(y, x_1, \ldots, x_n)$ is a function then the μ-minimalisation $g(x_1, \ldots, x_n) = \mu y\,[f(y, x_1, \ldots, x_n)]$ is the first value y such that $f(y, x_1, \ldots, x_n) = 0$. The function g can be partial, since f might at certain combinations of $x_1, \ldots, x_n$ not take the value 0 for any y and then the search for the y is undefined.*

The partial recursive or μ-recursive functions are those which are formed from the base functions by concatenation, primitive recursion and μ-minimalisation. If a partial recursive function is total, it is just called a recursive function.

Theorem 15.22. *Every partial recursive function can be computed by a register machine.*

Proof. It is easy to see that all base functions are computed by register machines and also the concatenation of functions. For the primitive recursion, one uses subprograms F for f and G for g. Now h is computed by the following program H. For simplicity, assume that f has two and g has four inputs.

Line 1: Function H(R_1, R_2, R_3);
Line 2: $R_4 = 0$;
Line 3: $R_5 = \mathrm{F}(R_2, R_3)$;
Line 4: If $R_4 = R_1$ Then Goto Line 8;
Line 5: $R_5 = \mathrm{G}(R_4, R_5, R_2, R_3)$;
Line 6: $R_4 = R_4 + 1$;

Line 7: Goto Line 4;
Line 8: Return(R_5).

Furthermore, the μ-minimalisation h of a function f can be implemented using a subprogram F for f; here one assumes that f has three and g two inputs.

Line 1: Function H(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = \mathrm{F}(R_3, R_1, R_2)$;
Line 4: If $R_4 = 0$ Then Goto Line 7;
Line 5: $R_3 = R_3 + 1$;
Line 6: Goto Line 3;
Line 7: Return(R_3).

These arguments show that whenever the given functions f, g can be computed by register programs, so are the functions derived from f and g by primitive recursion or μ-minimalisation. Together with the corresponding result for concatenation, one can derive that all partial recursive functions can be computed by register programs. ∎

Indeed, one can also show the converse direction that all partial functions computed by register programs are partial recursive. Thus one gets the following equivalence.

Theorem 15.23. *For a partial function f, the following are equivalent:*

- *f as a function from strings to strings can be computed by a Turing machine;*
- *f as a function from natural numbers to natural numbers can be computed by a register machine;*
- *f as a function from natural numbers to natural numbers is partial recursive.*

Alonzo Church formulated the following thesis which is also a basic assumption of Turing's work on the Entscheidungsproblem [76]; therefore the thesis is also known as "Church–Turing Thesis".

Thesis 15.24: Church's Thesis. *All sufficiently reasonable models of computation on $\mathbb{N}$ or on Σ^* are equivalent and give the same class of functions.*

One can also use Turing machines to define notions from complexity theory like classes of time usage or space usage. The time used by a Turing machine is the number of steps it makes until it halts; the space used is the number of different cells on the tape the head visits during a computation. One measures the size n of the input x in the number of its symbols in the language model and by $\log(x) = \min\{n \in \mathbb{N} : x \leq 2^n\}$ in the numerical model.

Theorem 15.25. *A function f is computable by a Turing machine in time $p(n)$ for some polynomial p iff f is computable by a register machine in time $q(n)$ for some polynomial q.*

Theorem 15.26. *A function f is computable by a Turing machine in space $p(n)$ for some polynomial p iff f is computable by a register machine in such a way that all registers take at most the value $2^{q(n)}$ for some polynomial q.*

The notions in Complexity Theory are also relatively invariant against changes of the model of computation; however, one has to interpret the word "reasonable" of Church in a stronger way than in recursion theory. Note that for these purposes, the model of a register machine where it can only count up steps of one is not reasonable, as then even the function $x \mapsto x + x$ is not computed in polynomial time. On the other hand, a model where the multiplication is also a primitive, one step operation, would also be unreasonable as then single steps have too much power. However, multiplication is still in polynomial time.

Example 15.27. The following register program computes multiplication in polynomial time.

Line 1: Function Polymult(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;
Line 4: If $R_3 = R_1$ Then Goto Line 13;
Line 5: $R_5 = 1$;
Line 6: $R_6 = R_2$;
Line 7: If $R_3 + R_5 > R_1$ Then Goto Line 4;
Line 8: $R_3 = R_3 + R_5$;
Line 9: $R_4 = R_4 + R_6$;
Line 10: $R_5 = R_5 + R_5$;
Line 11: $R_6 = R_6 + R_6$;
Line 12: Goto Line 7;
Line 13: Return(R_4).

Exercise 15.28. *Write a register program which computes the remainder in polynomial time.*

Exercise 15.29. *Write a register program which divides in polynomial time.*

Exercise 15.30. *Let an extended register machine have the additional command which permits to multiply two registers in one step. Show that an extended register* *machine can compute a function in polynomial time which cannot be computed in polynomial time by a normal register machine.*

16 Recursively Enumerable Sets

A special form of programs can employ For-Loops in place of arbitrary Goto-commands. Such a register machine program does not use backward goto-commands except for a For-Loop which has to satisfy the following condition: The Loop variables does not get changed inside the loop and the bounds are read out when entering the loop and not changed during the run of the loop. For-Loops can be nested but they cannot partly overlap. Goto commands can neither go into a loop nor from inside a loop out of it. The rules for Goto commands also apply for if-commands. Here an example.

Line 1: Function Factor(R_1, R_2);
Line 2: $R_3 = R_1$;
Line 3: $R_4 = 0$;
Line 4: If $R_2 < 2$ Then Goto Line 10;
Line 5: For $R_5 = 0$ to R_1
Line 6: If Remainder$(R_3, R_2) > 0$ Then Goto Line 9;
Line 7: $R_3 = \text{Divide}(R_3, R_2)$;
Line 8: $R_4 = R_4 + 1$;
Line 9: Next R_5;
Line 10: Return(R_4);

This function computes how often R_2 is a factor of R_1 and is primitive recursive. Using For-Loops to show that programs are primitive recursive is easier than to follow the scheme of primitive recursion precisely. Consider the following easy function.

Line 1: Function Collatz(R_1);
Line 2: If Remainder$(R_1, 2) = 0$ Then Goto Line 6;
Line 3: If $R_1 = 1$ Then Goto Line 8;
Line 4: $R_1 = \text{Mult}(R_1, 3) + 1$;
Line 5: Goto Line 2;
Line 6: $R_1 = \text{Divide}(R_1, 2)$;
Line 7: Goto Line 2;
Line 8: Return(R_1);

It is unknown whether this function terminates for all inputs larger than 1. Lothar Collatz conjectured in 1937 that "yes", but though many attempts have been made since then, no proof has been found that termination is there. Though one does not know whether the function terminates on a particular output, one can write a function which simulates "Collatz" for R_2 steps with a For-Loop. In the case that the output line is reached with output y, one outputs $y + 1$; in the case that the output line is not reached, one outputs 0. This simulating function is primitive recursive.

In order of avoiding too much hard coding in the function, several instructions per line are allowed. The register LN for the line number and T for the loop are made explicit.

Line 1: Function Collatz(R_1, R_2);
Line 2: $LN = 2$;
Line 3: For $T = 0$ to R_2
Line 4: If $LN = 2$ Then Begin If Remainder$(R_1, 2) = 0$ Then $LN = 6$ Else $LN = 3$; Goto Line 10 End;
Line 5: If $LN = 3$ Then Begin If $R_1 = 1$ Then $LN = 8$ Else $LN = 4$; Goto Line 10 End;
Line 6: If $LN = 4$ Then Begin $R_1 = \text{Mult}(R_1, 3) + 1$; $LN = 5$; Goto Line 10 End;
Line 7: If $LN = 5$ Then Begin $LN = 2$; Goto Line 10 End;
Line 8: If $LN = 6$ Then Begin $R_1 = \text{Divide}(R_1, 2)$; $LN = 7$; Goto Line 10 End;
Line 9: If $LN = 7$ Then Begin $LN = 2$; Goto Line 10 End;
Line 10: Next T;
Line 11: If $LN = 8$ Then Return$(R_1 + 1)$ Else Return(0);

In short words, in every simulation step, the action belonging to the line number LN is carried out and the line number is afterwards updated accordingly. The simulation here is not yet perfect, as "Mult" and "Divide" are simulated in one step; a more honest simulation would replace this macros by the basic commands and then carry out the simulation.

Exercise 16.1. *Write a program for a primitive recursive function which simulate the following function with input R_1 for R_2 steps.*

Line 1: Function Expo(R_1);
Line 2: $R_3 = 1$;
Line 3: If $R_1 = 0$ Then Goto Line 7;
Line 4: $R_3 = R_3 + R_3$;
Line 5: $R_1 = R_1 - 1$;
Line 6: Goto Line 3;
Line 7: Return(R_3).

Exercise 16.2. *Write a program for a primitive recursive function which simulate the following function with input R_1 for R_2 steps.*

Line 1: Function Repeatadd(R_1);
Line 2: $R_3 = 3$;
Line 3: If $R_1 = 0$ Then Goto Line 7;
Line 4: $R_3 = R_3 + R_3 + R_3 + 3$;
Line 5: $R_1 = R_1 - 1$;
Line 6: Goto Line 3;
Line 7: Return(R_3).

Theorem 16.3. *For every partial-recursive function f there is a primitive recursive function g and a register machine M such that for all t,*

If $f(x_1, \ldots, x_n)$ is computed by M within t steps
Then $g(x_1, \ldots, x_n, t) = f(x_1, \ldots, x_n) + 1$
Else $g(x_1, \ldots, x_n, t) = 0$.

In short words, g simulates the program M of f for t steps and if an output y comes then g outputs $y + 1$ else g outputs 0.

Based on Theorem 16.3, one can make many equivalent formalisations for the notion that a set is enumerated by an algorithm.

Theorem 16.4. *The following notions are equivalent for a set $A \subseteq \mathbb{N}$:*

(a) *A is the range of a partial recursive function;*
(b) *A is empty or A is the range of a total recursive function;*
(c) *A is empty or A is the range of a primitive recursive function;*
(d) *A is the set of inputs on which some register machine terminates;*
(e) *A is the domain of a partial recursive function;*
(f) *There is a two-place recursive function g such that $A = \{x : \exists y\, [g(x, y) > 1]\}$.*

Proof. $(a) \Rightarrow (c)$: If A is empty then (c) holds; if A is not empty then there is an element $a \in A$ which is now taken as a constant. For the partial function f whose range A is, there is, by Theorem 16.3, a primitive function g such that either $g(x, t) = 0$ or $g(x, t) = f(x) + 1$ and whenever $f(x)$ takes a value there is also a t with $g(x, t) = f(x) + 1$. Now one defines a new function h which is also primitive recursive such that if $g(x, t) = 0$ then $h(x, t) = a$ else $h(x, t) = g(x, t) - 1$. The range of h is A.

$(c) \Rightarrow (b)$: This follows by definition as every primitive recursive function is also recursive.

$(b) \Rightarrow (d)$: Given a function h whose range is A, one can make a register machine which simulates h and searches over all possible inputs and checks whether h on these inputs is x. If such inputs are found then the search terminates else the register machine runs forever. Thus $x \in A$ iff the register machine program following this behaviour terminates after some time.

$(d) \Rightarrow (e)$: The domain of a register machine is the set of inputs on which it halts and outputs a return value. Thus this implication is satisfied trivially by taking the function for (e) to be exactly the function computed from the register program for (d).

$(e) \Rightarrow (f)$: Given a register program f whose domain A is according to (e), one takes the function g as defined by Theorem 16.3 and this function indeed satisfies that $f(x)$ is defined iff there is a t such that $g(x, t) > 0$.

$(f) \Rightarrow (a)$: Given the function g as defined in (f), one defines that if there is a t with $g(x, t) > 0$ then $f(x) = x$ else $f(x)$ is undefined. The latter comes by infinite search for a t which is not found. Thus the partial recursive function f has range A. ∎

The many equivalent definitions show that they capture a natural concept. This is formalised in the following definition (which could take any of the above entries).

Definition 16.5. *A set is recursively enumerable iff it is the range of a partial recursive function.*

If a set is recursively enumerable there is a function which can enumerate the members; however, often one wants the better property to decide the membership in the set. This property is defined as follows.

Definition 16.6. *A set $L \subseteq \mathbb{N}$ is called recursive or decidable iff the function $x \mapsto L(x)$ with $L(x) = 1$ for $x \in L$ and $L(x) = 0$ for $x \notin L$ is recursive; L is undecidable or nonrecursive iff this function is not recursive, that is, if there is no algorithm which can decide whether $x \in L$.*

One can also cast the same definition in the symbolic model. Let Σ be a finite alphabet and $A \subseteq \Sigma^*$. The set A is recursively enumerable iff it is the range of a partial function computed by a Turing machine and A is recursive or decidable iff the mapping $x \mapsto A(x)$ is computed by a Turing machine. There is a natural characterisation. The next result shows that not all recursively enumerable sets are decidable. The most famous example is due to Turing.

Definition 16.7: Halting Problem [76]. Let $e, x \mapsto \varphi_e(x)$ be a universal partial recursive function covering all one-variable partial recursive functions. Then the set $H = \{(e, x) : \varphi_e(x) \text{ is defined}\}$ is called the *general halting problem* and $K = \{e : \varphi_e(e)\}$ is called the *diagonal halting problem*.

The name stems from the fact that Turing considered universal partial recursive functions which are defined using Turing machines or register machines or any other such natural mechanism. Then $\varphi_e(x)$ is defined iff the e-th register machine with input x halts and produces some output.

Theorem 16.8: Undecidability of the Halting Problem [76]. *Both the diagonal halting problem and the general halting problem are recursively enumerable and undecidable.*

Proof. It is sufficient to prove this for the diagonal halting problem. Note that Turing [76] proved that a universal function like $e, x \mapsto \varphi_e(x)$ exists, that is, that one can construct a partial recursive function which simulates on input e and x the behaviour of the e-th register machine with one input. Let $\mathrm{F}(e, x)$ be this function. Furthermore let $\mathrm{Halt}(e)$ be a program which checks whether $\varphi_e(e)$ halts; it outputs 1 in the case of "yes" and 0 in the case of "no". Now one can make the following register program using F and Halt as macros.

Line 1: Function Diagonalise(R_1);
Line 2: $R_2 = 0$;
Line 3: If $\mathrm{Halt}(R_1) = 0$ Then Goto Line 5;
Line 4: $R_2 = \mathrm{F}(R_1, R_1) + 1$;
Line 5: Return(R_2).

Note that Diagonalise is a total function: On input e it first checks whether $\varphi_e(e)$ is defined using Halt. If not, Diagonalise(e) is 0 and therefore different from $\varphi_e(e)$ which is undefined.l If yes, Diagonalise(e) is $\varphi_e(e) + 1$, as $\varphi_e(e)$ can be computed by doing the simulation $\mathrm{F}(e, e)$ and then adding one to it. So one can see that for all e, the function φ_e differs from Diagonalise on input e. Thus Diagonalise is a register machine having a different input/output behaviour than all the functions φ_e. Thus there are three possibilities to explain this:

1. The list $\varphi_0, \varphi_1, \ldots$ captures only some but not all functions computed by register machines;
2. The simulation $\mathrm{F}(e, e)$ to compute $\varphi_e(e)$ cannot be implemented;
3. The function $\mathrm{Halt}(e)$ does not always work properly, for example, it might on some inputs not terminate with an output.

The first two items — that register machines cover all partial-recursive functions and that the universal simulating register machine / partial recursive function exists — has been proven before by many authors and is correct. Thus the third assumption, that the function Halt exists and is total and does what it promises, must be the failure. This gives then Turing's result on the unsolvability of the halting problem.

The halting problem is recursively enumerable — see Entry **(d)** in Theorem 16.4 and the fact that there is a register machine computing $e \mapsto \varphi_e(e)$ — and therefore it is an example of a recursively enumerable set which is undecidable. This notion is formalised in the following definition. ∎

There is a close connection between recursively enumerable and recursive sets.

Theorem 16.9. *A set L is recursive iff both L and $\mathbb{N} - L$ are recursively enumerable.*

Exercise 16.10. *Prove this characterisation.*

Exercise 16.11. *Prove that the set $\{e : \varphi_e(2e+5)$ is defined$\}$ is undecidable.*

Exercise 16.12. *Prove that the set $\{e : \varphi_e(e^2+1)$ is defined$\}$ is undecidable.*

Exercise 16.13. *Prove that the set $\{e : \varphi_e(e/2)$ is defined$\}$ is undecidable. Here $e/2$ is the downrounded value of e divided by 2, so $1/2$ should be 0 and $3/2$ should be 1.*

Exercise 16.14. *Prove that the set $\{x^2 : x \in \mathbb{N}\}$ is recursively enumerable by proving that there is a register machine which halts exactly when a number is square.*

Exercise 16.15. *Prove that the set of prime numbers is recursively enumerable by proving that there is a register machine which halts exactly when a number is prime.*

Exercise 16.16. *Prove that the set $\{e : \varphi_e(e/2)$ is defined$\}$ is recursively enumerable by proving that it is the range of a primitive recursive function. Here $e/2$ is the downrounded value of e divided by 2, so $1/2$ should be 0 and $3/2$ should be 1.*

17 Undecidable Problems

Hilbert posed in the year 1900 in total 23 famous open problems. One of them was the task to construct an algorithm to determine the members of a Diophantine set. Among them are Diophantine sets. These sets can be defined using polynomials, either over the integers $\mathbb{Z}$ or over the natural numbers $\mathbb{N}$. Let $P(B)$ be the set of all polynomials with coefficients from B, for example, if $B = \{0, 1, 2\}$ then $P(B)$ contains polynomials like $1 \cdot x_1 + 2 \cdot x_2 x_3 + 1 \cdot x_3^5$.

Definition 17.1. *$A \subseteq \mathbb{N}$ is Diophantine iff one of the following equivalent conditions are true:*

(a) *There are n and a polynomials $p(x, y_1, \ldots, y_n), q(x, y_1, \ldots, y_n) \in P(\mathbb{N})$ such that, for all $x \in \mathbb{N}$,*

$$x \in A \Leftrightarrow \exists y_1, \ldots, y_n \in \mathbb{N}\, [p(x, y_1, \ldots, y_n) = q(x, y_1, \ldots, y_n)];$$

(b) *There are n and a polynomial $p(x, y_1, \ldots, y_n) \in P(\mathbb{Z})$ such that, for all $x \in \mathbb{N}$,*

$$x \in A \Leftrightarrow \exists y_1, \ldots, y_n \in \mathbb{N}\, [p(x, y_1, \ldots, y_n) = 0];$$

(c) *There are n and a polynomial $p(x, y_1, \ldots, y_n) \in P(\mathbb{Z})$ such that, for all $x \in \mathbb{N}$,*

$$x \in A \Leftrightarrow \exists y_1, \ldots, y_n \in \mathbb{Z}\, [p(x, y_1, \ldots, y_n) = 0];$$

(d) *There are n and a polynomial $p(y_1, \ldots, y_n) \in P(\mathbb{Z})$ such that, for all $x \in \mathbb{N}$,*

$$x \in A \Leftrightarrow \exists y_1, \ldots, y_n \in \mathbb{Z}\, [p(y_1, \ldots, y_n) = x],$$

that is, A is the intersection of $\mathbb{N}$ and the range of p.

Proposition 17.2. *The conditions* **(a)** *through* **(d)** *in Definition 17.1 are indeed all equivalent.*

Proof. (**a**) $\Rightarrow$ (**b**): The functions p, q from condition (**a**) have natural numbers as coefficients; their difference has integers as coefficients and $(p - q)(x, y_1, \ldots, y_n) = 0 \Leftrightarrow p(x, y_1, \ldots, y_n) = q(x, y_1, \ldots, y_n)$.

(**b**) $\Rightarrow$ (**c**): The functions p from condition (**b**) is of the corresponding form, however, the variables have to be quantified over natural numbers in **(b)** while over integers in **(c)**. The way out is to use the following result from number theory: Every natural number is the sum of four squares of natural numbers; for example, $6 = 0 + 1 + 1 + 4$. Furthermore, as squares of integers are always in $\mathbb{N}$, their sum is as well. So one can write

There are n and a polynomial $p(x, y_1, \ldots, y_n) \in P(\mathbb{Z})$ such that, for all $x \in \mathbb{N}$, $x \in A$ iff

$\exists z_1, \ldots, z_{4n} \in \mathbb{Z}\,[p(x, z_1^2+z_2^2+z_3^2+z_4^2, \ldots, z_{4n-3}^2+z_{4n-2}^2+z_{4n-1}^2+z_{4n}^2) = 0]$.

Thus the function $q(x, z_1, \ldots, z_{4n})$ given as $p(x, z_1^2 + z_2^2 + z_3^2 + z_4^2, \ldots, z_{4n-3}^2 + z_{4n-2}^2 + z_{4n-1}^2 + z_{4n}^2)$ is then the polynomial which is sought for in **(c)**.

(**c**) $\Rightarrow$ (**d**): The functions p from condition **(c)** can be used to make the corresponding condition for **(d)**. Indeed, if $p(x, y_1, \ldots, y_n) = 0$ then it follows that

$$q(x, y_1, \ldots, y_n) = x - (x+1) \cdot (p(x, y_1, \ldots, y_n))^2$$

takes the value x in the case that $p(x, y_1, \ldots, y_n) = 0$ and takes a negative number as value in the case that the polynomial $p(x, y_1, \ldots, y_n)$ has the absolute value of at least 1 and therefore also the square $(p(x, y_1, \ldots, y_n))^2$ has at least the value 1. Thus q can be used as the polynomial in **(d)**.

(**d**) $\Rightarrow$ (**a**): The functions p from condition **(d)** can be modified to match condition **(a)** in three steps: First one replaces each input y_k by $z_{2k-1} - z_{2k}$ where z_{2k-1}, z_{2k} are variables ranging over $\mathbb{N}$. Second one forms the polynomial

$$(x - p(z_1 - z_2, z_3 - z_4, \ldots, z_{2n-1} - z_{2n}))^2$$

which takes as values only natural numbers and has as variables only natural numbers. Now any polynomial equation mapping to 0 like

$$x^2 - 4xz_1 + 4xz_2 + 4z_1^2 + 4z_2^2 - 8z_1z_2 = 0$$

can be transformed to the equality of two members of $P(\mathbb{N})$ by brining terms with negative coefficient onto the other side:

$$x^2 + 4xz_2 + 4z_1^2 + 4z_2^2 = 4xz_1 + 8z_1z_2.$$

This then permits to choose the polynomials for **(a)**. ∎

Example 17.3. The set of all composite numbers (which are the product of at least two prime numbers) is Diophantine. So x is composite iff

$$x = (2 + y_1^2 + y_2^2 + y_3^2 + y_4^2) \cdot (2 + y_5^2 + y_6^2 + y_7^2 + y_8^2)$$

for some $y_1, \ldots, y_8 \in \mathbb{Z}$. Thus condition **(d)** shows that the set is Diophantine. The set of all square numbers is Diophantine: x is a square iff $x = y_1^2$ for some y_1.

The set of all non-square numbers is Diophantine. Here one could use condition **(b)** best and show that x is a non-square iff

$$\exists y_1, y_2, y_3 \in \mathbb{N}\,[x = y_1^2 + 1 + y_2 \text{ and } x + y_3 = y_1^2 + 2y_1]$$

which is equivalent to

$$\exists y_1, y_2, y_3 \in \mathbb{N}\,[(y_1^2 + 1 + y_2 - x)^2 + (x + y_3 - y_1^2 - 2y_1)^2 = 0].$$

This second condition has now the form of **(b)** and it says that x is properly between y_1^2 and $(y_1 + 1)^2$ for some y_1.

Quiz

(a) *Which numbers are in the Diophantine set* $\{x : \exists y \in \mathbb{N}\,[x = 4 \cdot y + 2]\}$*?*
(b) *Which numbers are in the Diophantine set* $\{x : \exists y \in \mathbb{N}\,[x^{16} = 17 \cdot y + 1]\}$*?*

Exercise 17.4. *Show that the set of all* $x \in \mathbb{N}$ *such that* x *is odd and* x *is a multiple of* 97 *is Diophantine.*

Exercise 17.5. *Show that the set of all natural numbers which are multiples of* 5 *but not multiples of* 7 *is Diophantine.*

Exercise 17.6. *Consider the set*

$$\{x \in \mathbb{N} : \exists y_1, y_2 \in \mathbb{N}\,[((2y_1 + 3) \cdot y_2) - x = 0]\}.$$

This set is Diophantine by condition **(b)**. *Give a verbal description for this set.*

Proposition 17.7. *Every Diophantine set is recursively enumerable.*

Proof. If A is Diophantine and empty, it is clearly recursively enumerable. If A is Diophantine and non-empty, consider any $a \in A$. Furthermore, there is a polynomial $p(x, y_1, \ldots, y_n)$ in $P(\mathbb{Z})$ such that $x \in A$ iff there are $y_1, \ldots, y_n \in \mathbb{N}$ with $p(x, y_1, \ldots, y_n) = 0$. One can now easily build a register machine which does the following on input $x, y_1, \ldots, y_n$: If $p(x, y_1, \ldots, y_n) = 0$ then the register machine outputs x else the register machine outputs a. Thus A is the range of a total function computed by a register machine, that is, A is the range of a recursive function. It follows that A is recursively enumerable. ∎

Proposition 17.8. *If* A, B *are Diophantine sets so are* $A \cup B$ *and* $A \cap B$.

Proof. There are an n, m and polynomials p, q in $P(\mathbb{Z})$ such that

$$x \in A \Leftrightarrow \exists y_1, \ldots, y_n \in \mathbb{N}\,[p(x, y_1, \ldots, y_n) = 0]$$

and

$$x \in B \Leftrightarrow \exists z_1, \ldots, z_m \in \mathbb{N}\,[q(x, z_1, \ldots, z_m) = 0]$$

These two conditions can be combined. Now x is in $A \cup B$ iff $p(x, y_1, \ldots, y_n) \cdot q(x, z_1, \ldots, z_m) = 0$ for some $y_1, \ldots, y_n, z_1, \ldots, z_m \in \mathbb{N}$; the reason is that the product is 0 iff one of the factors is 0. Furthermore, $x \in A \cap B$ iff $(p(x, y_1, \ldots, y_n))^2 + (q(x, z_1, \ldots, z_m))^2 = 0$ for some $y_1, \ldots, y_n, z_1, \ldots, z_m \in \mathbb{N}$; the reason is that this sum is 0 iff both subpolynomials p, q evaluate to 0, that is, x is in both sets; note that the variables to be quantified over are different and therefore one can choose them independently from each other in order to get both of p, q to be 0 in the case that $x \in A \cap B$. ▮

Exercise 17.9. *Show that if a set A is Diophantine then also the set*

$$B = \{x \in \mathbb{N} : \exists x' \in \mathbb{N}\,[(x + x')^2 + x \in A]\}$$

is Diophantine.

David Hilbert asked in 1900 in an address to the International Congress of Mathematicians for an algorithm to determine whether Diophantine sets have members and to check whether a specific x would be a member of a Diophantine set; this was the tenth of his list of 23 problems he thought should be solved within the twentieth century. It turned out that this is impossible. In the 1930ies, mathematicians showed that there are recursively enumerable sets for which the membership cannot be decided, among them Alan Turing's halting problem to be the most famous one. In 1970, Matiyasevich [50, 51] showed that recursively enumerable subsets of $\mathbb{N}$ are Diophantine and thus there is no algorithm which can check whether a given x is a member of a given Diophantine set; even if one keeps the Diophantine set fixed.

Theorem 17.10: Unsolvability of Hilbert's Tenth Problem [50]**.** *Every recursively enumerable set is Diophantine; in particular there are undecidable Diophantine sets.*

A general question investigated by mathematicians is also how to decide the correctness of formulas which are more general than those defining Diophantine sets, that is, of formulas which also allow universal quantification. Such lead to the definition of arithmetic sets.

Definition 17.11. Arithmetic setsTa36 A set $A \subseteq \mathbb{N}$ is called *arithmetic* iff there is a formula using existential ($\exists$) and universal ($\forall$) quantifiers over variables such that all variables except for x are quantified and that the predicate behind the quantifiers only uses Boolean combinations of polynomials from $P(\mathbb{N})$ compared by $<$ and $=$ in order to evaluate the formula; formulas can have constants denoting the corresponding natural numbers, constants for 0 and 1 are sufficient.

The following examples are the starting point towards the undecidability of certain arithmetic sets.

Example 17.12. The set P of all prime numbers is defined by

$$x \in P \Leftrightarrow \forall y, z\, [x > 1 \text{ and } (y+2) \cdot (z+2) \neq x]$$

and the set T of all powers of 2 is defined by

$$x \in T \Leftrightarrow \forall y, y'\, \exists z\, [x > 0 \text{ and } (x = y \cdot y' \Rightarrow (y = 1 \text{ or } y = 2 \cdot z))]$$

and, in general, the set E of all prime powers is defined by

$$(p, x) \in E \Leftrightarrow \forall y, y'\, \exists z\, [p > 1 \text{ and } x \geq p \text{ and } (x = y \cdot y' \Rightarrow (y = 1 \text{ or } y = p \cdot z))]$$

which says that $(p, x) \in E$ iff p is a prime number and x is a non-zero power of p. In the last equations, E is a subset of $\mathbb{N} \times \mathbb{N}$ rather than $\mathbb{N}$ itself.

Example 17.13: Configuration and Update of a Register Machine [76]**.** The configuration of a register machine at step t is the line number LN of the line to be processed and the content $R_1, \ldots, R_n$ of the n registers. There is a set U of updates of tuples of the form $(LN, R_1, \ldots, R_n, LN', R'_1, \ldots, R'_n, p)$ where such a tuple is in U iff p is an upper bound on all the components in the tuple and the register program when being in line number LN and having the register content $R_1, \ldots, R_n$ goes in one step to line number LN' and has the content $R'_1, \ldots, R'_n$. Note that here upper bound means "strict upper bound", that is, $LN < p$ and $R_1 < p$ and $\ldots$ and $R_n < p$ and $LN' < p$ and $R'_1 < p$ and $\ldots$ and $R'_n < p$. Consider the following example program (which is a bit compressed to give an easier formula):

Line 1: Function Sum(R_1); $R_2 = 0$; $R_3 = 0$;
Line 2: $R_2 = R_2 + R_3$; $R_3 = R_3 + 1$;
Line 3: If $R_3 \leq R_1$ Then Goto Line 2;
Line 4: Return(R_2);

The set U would now be defined as follows:

$(LN, R_1, R_2, R_3, LN', R_1', R_2', R_3', p)$ is in U iff
$LN < p$ and $R_1 < p$ and $R_2 < p$ and $R_3 < p$ and $LN' < p$ and $R_1' < p$ and $R_2' < p$ and $R_3' < p$ and
$[(LN = 1$ and $LN' = 2$ and $R_1' = R_1$ and $R_2' = 0$ and $R_3' = 0)$ or $(LN = 2$ and $LN' = 3$ and $R_1' = R_1$ and $R_2' = R_2 + R_3$ and $R_3' = R_3 + 1)$ or
$(LN = 3$ and $LN' = 2$ and $R_1' = R_3'$ and $R_2' = R_2$ and $R_3' = R_3$ and $R_3 \leq R_1)$ or
$(LN = 3$ and $LN' = 4$ and $R_1' = R_3'$ and $R_2' = R_2$ and $R_3' = R_3$ and $R_3 > R_1)]$.

Note the longer the program and the more lines it has, the more complex are the update conditions. They have not only to specify which variables change, but also those which keep their values. Such an U can be defined for every register machine.

Example 17.14: Run of a Register Machine. One could code the values of the registers in digits step by step. For example, when all values are bounded by 10, for computing sum(3), the following sequences would permit to keep track of the configurations at each step:

```
LN: 1 2 3 2 3 2 3 2 3 4
R1: 3 3 3 3 3 3 3 3 3 3
R2: 0 0 0 0 1 1 3 3 6 6
R3: 0 0 1 1 2 2 3 3 4 4
```

So the third column says that after two steps, the register machine is going to do Line 3 and has register values 3,0,1 prior to doing the commands in Line 3. The last column says that the register machine has reached Line 4 and has register values 3,6,4 prior to doing the activity in Line 4 which is to give the output 6 of Register R_2 and terminate.

Now one could code each of these in a decimal number. The digit relating to 10^t would have the configurations of the registers and line numbers at the beginning of step t of the computation. Thus the corresponding decimal numbers would be 4323232321 for LN and 3333333333 for R_1 and 6633110000 for R_2 and 4433221100 for R_3. Note that the updates of a line take effect whenever the next step is executed.

In the real coding, one would not use 10 but a prime number p. The value of this prime number p just depends on the values the registers take during the computation; the larger these are, the larger p has to be. Now the idea is that the p-adic digits for p^t code the values at step t and for p^{t+1} code the values at step $t + 1$ so that one can check the update.

Now one can say that the program Sum(x) computes the value y iff there exist q, p, LN, R_1, R_2, R_3 such that q is a power of p and p is a prime and LN, R_1, R_2, R_3

code a run with input x and output y in the format given by p, q. More precisely, for given x, y there have to exist p, q, LN, R_1, R_2, R_3 satisfying the following conditions:

1. $(p, q) \in E$, that is, p is a prime and q is a power of p;
2. $R_1 = r_1 \cdot p + x$ and $LN = r_{LN} \cdot p + 1$ and $p > x + 1$ for some numbers r_{LN}, r_1;
3. $R_2 = q \cdot y + r_2$ and $LN = q \cdot 4 + r_{LN}$ and $p > y + 4$ for some numbers $r_2, r_{LN} < q$;
4. For each $p' < q$ such that p' divides q there are $r_{LN}, r_1, r_2, r_3, r'_{LN}, r'_1, r'_2, r'_3, r''_{LN}, r''_1, r''_2, r''_3, r'''_{LN}, r'''_1, r'''_2, r'''_3$ such that
   - $r_{LN} < p'$ and $LN = r_{LN} + p' \cdot r'_{LN} + p' \cdot p \cdot r''_{LN} + p' \cdot p^2 \cdot r'''_{LN}$;
   - $r_1 < p'$ and $R_1 = r_1 + p' \cdot r'_1 + p' \cdot p \cdot r''_1 + p' \cdot p^2 \cdot r'''_1$;
   - $r_2 < p'$ and $R_2 = r_2 + p' \cdot r'_2 + p' \cdot p \cdot r''_2 + p' \cdot p^2 \cdot r'''_2$;
   - $r_3 < p'$ and $R_3 = r_3 + p' \cdot r'_3 + p' \cdot p \cdot r''_3 + p' \cdot p^2 \cdot r'''_3$;
   - $(r'_{LN}, r'_1, r'_2, r'_3, r''_{LN}, r''_1, r''_2, r''_3, p) \in U$.

This can be formalised by a set R of pairs of numbers such that $(x, y) \in R$ iff the above described quantified formula is true. Thus there is a formula in arithmetics on $(\mathbb{N}, +, \cdot)$ using both types of quantifier $(\exists, \forall)$ which is true iff the register machine computes from input x the output y.

Furthermore, one can also define when this register machine halts on input x by saying that the machine halts on x iff $\exists y\, [(x, y) \in R]$.

This can be generalised to any register machine computation including one which simulates on input e, x the e-th register machine with input x (or the e-th Turing machine with input x). Thus there is a set H definable in arithmetic on the natural numbers such that $(e, x) \in H$ iff the e-th register machine with input x halts. This gives the following result of Turing.

Theorem 17.15: Undecidability of Arithmetics. The set of all true formulas in arithmetic of the natural numbers with $+$ and $\cdot$ using universal $(\forall)$ and existential $(\exists)$ quantification over variables is undecidable.

Church [16] and Turing [76] also used this construction to show that there is no general algorithm which can check for any logical formula, whether it is valid, that is, true in all logical structures having the operations used in the formula. Their work solved the Entscheidungsproblem of Hilbert from 1928. Note that the Entscheidungsproblem did not talk about a specific structure like the natural numbers. Instead Hilbert asked whether one can decide whether a logical formula is true in all structures to which the formula might apply; for example, whether a formula involving $+$ and $\cdot$ is true in all structures which have an addition and multiplication.

One might ask whether every arithmetical set is at least recursively enumerable.

The next results will show that this is not the case; for this one needs the following definition.

Definition 17.16. *A set $I \subseteq \mathbb{N}$ is an index set iff for all $d, e \in \mathbb{N}$, if $\varphi_d = \varphi_e$ then either d, e are both in I or d, e are both outside I.*

The definition of an index set has implicit the notion of the numbering on which it is based. For getting the intended results, one has to assume that the numbering has a certain property which is called "acceptable".

Definition 17.17: Acceptable Numbering [30]**.** For index sets, it is important to see on what numbering they are based. Here a numbering is a two-place function $e, x \mapsto \varphi_e(x)$ of functions φ_e having one input which is partial recursive (in both e and x). A numbering φ is acceptable iff for every further numbering ψ there is a recursive function f such that, for all e, $\psi_e = \varphi_{f(e)}$. That is, f translates "indices" or "programs" of ψ into "indices" or "programs" of φ which do the same.

The universal functions for register machines and for Turing machines considered above in these notes are actually acceptable numberings. The following proposition is more or less a restatement of the definition of acceptable.

Proposition 17.18. *Let φ be an acceptable numbering and f be a partial-recursive function with $n+1$ inputs. Then there is a recursive function g with n inputs such that*

$$\forall e_1, \ldots, e_n, x\,[f(e_1, \ldots, e_n, x) = \varphi_{g(e_1,\ldots,e_n)}(x)]$$

equality means that either both sides are defined and equal or both sides are undefined.

This proposition is helpful to prove the following theorem of Rice which is one of the milestones in the study of index sets and undecidable problems. The proposition is in that proof mainly used for the parameters $n = 1$ and $n = 2$. For the latter note that $e_1, e_2 \mapsto (e_1 + e_2) \cdot (e_1 + e_2 + 1)/2 + e_2$ is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ and it is easy to see that it is a recursive bijection, as it is a polynomial. Now given f with inputs e_1, e_2, x, one can make a numbering ψ defined by

$$\psi_{(e_1+e_2)\cdot(e_1+e_2+1)/2+e_2}(x) = f(e_1, e_2, x)$$

and then use that due to φ being acceptable there is a recursive function $\tilde{g}$ with

$$\psi_e = \varphi_{\tilde{g}(e)}$$

for all e. Now let $g(e_1, e_2) = \tilde{g}((e_1 + e_2) \cdot (e_1 + e_2 + 1)/2 + e_2)$ and it follows that

$$\forall e_1, e_2, x\,[f(e_1, e_2, x) = \varphi_{g(e_1,e_2)}(x)]$$

where, as usual, two functions are equal at given inputs if either both sides are defined and take the same value or both sides are undefined. The function g is the concatenation of the recursive function $\tilde{g}$ with a polynomial and thus recursive.

Theorem 17.19: Rice's Characterisation of Index Sets [63]. *Let φ be an acceptable numbering and I be an index set (with respect to φ).*

(a) *The set I is recursive iff $I = \emptyset$ or $I = \mathbb{N}$.*

(b) *The set I is recursively enumerable iff there is a recursive enumeration of finite lists $(x_1, y_1, \ldots, x_n, y_n)$ of conditions such that every index e satisfies that $e \in I$ iff there is a list $(x_1, y_1, \ldots, x_n, y_n)$ in the enumeration such that, for $m = 1, \ldots, n$, $\varphi_e(x_m)$ is defined and equal to y_m.*

Proof. First one looks into case **(b)** and assume that there is an enumeration of the lists $(x_1, y_1, \ldots, x_n, y_n)$ such that each partial function in I satisfies at least the conditions of one of these lists. Now one can define that $f(d, e)$ takes the value e in the case that $\varphi_e(x_1) = y_1, \ldots, \varphi_e(x_n) = y_n$ for the d-th list $(x_1, y_1, \ldots, x_n, y_n)$ in this enumeration; in the case that the d-th list has the parameter $n = 0$ (is empty) then $f(d, e) = e$ without any further check. In the case that the simulations for one x_m to compute $\varphi_e(x_m)$ does not terminate or gives a value different from y_m then $f(d, e)$ is undefined. Thus the index set I is range of a partial recursive function and therefore I is recursively enumerable.

Now assume for the converse direction that I is recursively enumerable. Let $Time(e, x)$ denote the time that a register machine needs to compute $\varphi_e(x)$; if this computation does not halt then $Time(e, x)$ is also undefined and considered to be ∞ so that $Time(e, x) > t$ for all $t \in \mathbb{N}$. Note that the set of all (e, x, t) with $Time(e, x) \leq t$ is recursive.

Now define $f(i, j, x)$ as follows: If $Time(i, x)$ is defined and it furthermore holds that $Time(j, j) > Time(i, x) + x$ then $f(i, j, x) = \varphi_i(x)$ else $f(i, j, x)$ remains undefined.

The function f is partial recursive. The function f does the following: if $\varphi_i(x)$ halts and furthermore $\varphi_j(j)$ does not halt within $Time(i, x) + x$ then $f(i, j, x) = \varphi_i(x)$ else $f(i, j, x)$ is undefined. By Proposition 17.18, there is a function g such that

$$\forall i, j, x \, [\varphi_{g(i,j)}(x) = f(i, j, x)]$$

where again equality holds if either both sides of the equality are defined and equal or both sides are undefined.

Now consider any $i \in I$. For all j with $\varphi_j(j)$ being undefined, it holds that

$$\varphi_i = \varphi_{g(i,j)}$$

and therefore $g(i,j) \in I$. The complement of the diagonal halting problem is not recursively enumerable while the set $\{j : g(i,j) \in I\}$ is recursively enumerable; thus there must be a j with $g(i,j) \in I$ and $\varphi_j(j)$ being defined. For this j, it holds that $\varphi_{g(i,j)}(x)$ is defined iff $Time(e,x) + x < Time(j,j)$. This condition can be tested effectively and the condition is not satisfied for any $x \geq Time(j,j)$. Thus one can compute an explicit list $(x_1, y_1, \ldots, x_n, y_n)$ such that $\varphi_{g(i,j)}(x)$ is defined and takes the value y iff there is an $m \in \{1, \ldots, n\}$ with $x = x_m$ and $y = y_m$. There is an algorithm which enumerates all these lists, that is, the set of these lists is recursively enumerable. This list satisfies therefore the following:

- If $i \in I$ then a there is a list $(x_1, y_1, \ldots, x_n, y_n)$ enumerated such that $\varphi_i(x_1) = y_1, \ldots, \varphi_i(x_n) = y_n$; note that this list might be empty ($n = 0$), for example in the case that φ_i is everywhere undefined;
- If $(x_1, y_1, \ldots, x_n, y_n)$ appears in the list then there is an index $i \in I$ such that $\varphi_i(x)$ is defined and equal to y iff there is an $m \in \{1, \ldots, n\}$ with $x_m = x$ and $y_m = y$.

What is missing is that all functions extending a tuple from the list have also their indices in I. So consider any tuple $(x_1, y_1, \ldots, x_n, y_n)$ in the list and any function φ_i extending this tuple. Now consider the following partial function f': $f'(j,x) = y$ iff either there is an $m \in \{1, \ldots, n\}$ with $x_m = x$ and $y_m = y$ or $\varphi_j(j)$ is defined and $\varphi_i(x) = y$. There is a recursive function g' with $\varphi_{g'(j)}(x) = f'(j,x)$ for all j, x; again either both sides of the equation are defined and equal or both sides are undefined. Now the set $\{j : g'(j) \in I\}$ is recursive enumerable and it contains all j with $\varphi_j(j)$ being undefined; as the diagonal halting problem is not recursive, the set $\{j : g'(j) \in I\}$ is a proper superset of $\{j : \varphi_j(j) \text{ is undefined}\}$. As there are only indices for two different functions in the range of g, it follows that $\{j : g'(j) \in I\} = \mathbb{N}$. Thus $i \in I$ and the set I coincides with the set of all indices e such that some finite list $(x_1, y_1, \ldots, x_n, y_n)$ is enumerated with $\varphi_e(x_m)$ being defined and equal to y_m for all $m \in \{1, \ldots, n\}$. This completes part **(b)**.

Second for the case **(a)**, it is obvious that $\emptyset$ and $\mathbb{N}$ are recursive index sets. So assume now that I is a recursive index set. Then both I and $\mathbb{N} - I$ are recursively enumerable. One of these sets, say I, contains an index e of the everywhere undefined function. By part **(b)**, the enumeration of conditions to describe the indices e in the index set I must contain the empty list. Then every index e satisfies the conditions in this list and therefore $I = \mathbb{N}$. Thus $\emptyset$ and $\mathbb{N}$ are the only two recursive index sets. ∎

Corollary 17.20. *There are arithmetic sets which are not recursively enumerable.*

Proof. Recall that the halting problem

$$H = \{(e,x) : \varphi_e(x) \text{ is defined}\}$$

is definable in arithmetic. Thus also the set

$$\{e : \forall x\, [(e, x) \in H]\}$$

of indices of all total functions is definable in arithmetic by adding one more quantifier to the definition, namely the universal one over all x. If this set would be recursively enumerable then there would recursive enumeration of lists of finite conditions such that when a function satisfies one list of conditions then it is in the index set. However, for each such list there is a function with finite domain satisfying it, hence the index set would contain an index of a function with a finite domain, in contradiction to its definition. Thus the set

$$\{e : \forall x\, [(e, x) \in H]\}$$

is not recursively enumerable. ▮

The proof of Rice's Theorem and also the above proof have implicitly used the following observation.

Observation 17.21. *If A, B are sets and B is recursively enumerable and if there is a recursive function g with $x \in A \Leftrightarrow g(x) \in B$ then A is also recursively enumerable.*

Such a function g is called a many-one reduction. Formally this is defined as follows.

Definition 17.22. *A set A is many-one reducible to a set B iff there is a recursive function g such that, for all x, $x \in A \Leftrightarrow g(x) \in B$.*

One can see from the definition: Assume that A is many-one reducible to B. If B is recursive so is A; if B is recursively enumerable so is A. Thus a common proof method to show that some set B is not recursive or not recursively enumerable is to find a many-one reduction from some set A to B where the set A is not recursive or recursively enumerable, respectively.

Example 17.23. The set $E = \{e : \forall \text{ even } x\, [\varphi_e(x) \text{ is defined}]\}$ is not recursively enumerable. This can be seen as follows: Define $f(e, x)$ such that $f(e, 2x) = f(e, 2x + 1) = \varphi_e(x)$ for all e, x. Now there is a recursive function g such that $\varphi_{g(e)}(x) = f(e, x)$ for all x; furthermore, $\varphi_{g(e)}(2x) = \varphi_e(x)$ for all e, x. It follows that φ_e is total iff $\varphi_{g(e)}$ is defined on all even inputs. Thus the set of all indices of total functions is many-one reducible to E via g and therefore E cannot be recursively enumerable.

Example 17.24. The set $F = \{e : \varphi_e \text{ is somewhere defined}\}$ is not recursive. There is a partial recursive function $f(e, x)$ with $f(e, x) = \varphi_e(e)$ for all e, x and a recursive function g with $\varphi_{g(e)}(x) = f(e,x) = \varphi_e(e)$ for all e. Now $e \in K$ iff $g(e) \in F$ and thus F is not recursive.

Theorem 17.25. *Every recursively enumerable set is many-one reducible to the diagonal halting problem $K = \{e : \varphi_e(e)$ is defined$\}$.*

Proof. Assume that A is recursively enumerable. Now there is a partial recursive function $\tilde{f}$ such that A is the domain of $\tilde{f}$. One adds to $\tilde{f}$ one input parameter which is ignored and obtains a function f such that $f(e,x)$ is defined iff $e \in A$. Now there is a recursive function g such that

$$\forall e,x\,[\varphi_{g(e)}(x) = f(e,x)].$$

If $e \in A$ then $\varphi_{g(e)}$ is total and $g(e) \in K$; if $e \notin A$ then $\varphi_{g(e)}$ is nowhere defined and $g(e) \notin K$. Thus g is a many-one reduction from A to K. ∎

Exercise 17.26. *Show that the set $F = \{e : \varphi_e$ is defined on at least one $x\}$ is many-one reducible to the set $\{e : \varphi_e(x)$ is defined for exactly one input $x\}$.*

Exercise 17.27. *Determine for the following set whether it is recursive, recursively enumerable and non-recursive or even not recursively enumerable: $A = \{e : \forall x\,[\varphi_e(x)$ is defined iff $\varphi_x(x+1)$ is undefined$]\}$.*

Exercise 17.28. *Determine for the following set whether it is recursive, recursively enumerable and non-recursive or even not recursively enumerable: $B = \{e :$ There are at least five numbers x where $\varphi_e(x)$ is defined$\}$.*

Exercise 17.29. *Determine for the following set whether it is recursive, recursively enumerable and non-recursive or even not recursively enumerable: $C = \{e :$ There are infinitely many x where $\varphi_e(x)$ is defined$\}$.*

Exercise 17.30. *Assume that φ_e is an acceptable numbering. Now define ψ such that*

$$\psi_{(d+e)\cdot(d+e+1)/2+e}(x) = \begin{cases} undefined & \text{if } d = 0 \text{ and } x = 0; \\ d-1 & \text{if } d > 0 \text{ and } x = 0; \\ \varphi_e(x) & \text{if } x > 0. \end{cases}$$

Is the numbering ψ enumerating all partial recursive functions? Is the numbering ψ an acceptable numbering?

Exercise 17.31. *Is there a numbering ϑ with the following properties:*

- *The set $\{e : \vartheta_e$ is total$\}$ is recursively enumerable;*
- *Every partial recursive function φ_e is equal to some ϑ_d.*

Prove the answer.

18 Undecidability and Formal Languages

The current section uses methods from the previous sections in order to show that certain problems in the area of formal languages are undecidable. Furthermore, this section adds another natural concept to describe recursively enumerable languages: they are those which are generated by some grammar. For the corresponding constructions, the notion of the register machine will be adjusted to the multi counter machine with respect to two major changes: the commands will be made much more simpler (so that computations / runs can easily be coded using grammars) and the numerical machine is adjusted to the setting of formal languages and reads the input in like a pushdown automaton (as opposed to register machines which have the input in some of the registers). There are one counter machines and multi counter machines; one counter machines are weaker than deterministic pushdown automata, therefore the natural concept is to allow several ("multi") counters.

Description 18.1: Multi Counter Automata. One can modify the pushdown automaton to counter automata, also called counter machines. Counter automata are like register machines and Turing machines controlled by line numbers or states (these concepts are isomorphic); the difference to register machines are the following two:

- The counters (= registers) have much more restricted operations: One can add or subtract 1 or compare whether they are 0. The initial values of all counters is 0.
- Like a pushdown automaton, one can read one symbol from the input at a time; depending on this symbol, the automaton can go to the corresponding line. One makes the additional rule that a run of the counter automaton is only valid iff the full input was read.
- The counter automaton can either output symbols with a special command (when computing a function) or terminate in lines with the special commands "ACCEPT" and "REJECT" in the case that no output is needed but just a binary decision. Running forever is also interpreted as rejection and in some cases it cannot be avoided that rejection is done this way.

Here an example of a counter automaton which reads inputs and checks whether at each stage of the run, at least as many 0 have been seen so far as 1.

Line 1: Counter Automaton Zeroone;
Line 2: Input Symbol – Symbol 0: Goto Line 3; Symbol 1: Goto Line 4; No further Input: Goto Line 7;
Line 3: $R_1 = R_1 + 1$; Goto Line 2;
Line 4: If $R_1 = 0$ Then Goto Line 6;

Line 5: $R_1 = R_1 - 1$; Goto Line 2;
Line 6: REJECT;
Line 7: ACCEPT.

A run of the automaton on input 001 would look like this:

```
Line:  1 2 3 2 3 2 4 5 2 7
Input:   0   0   1     -
R1:    0 0 0 1 1 2 2 2 1 1
```

A run of the automaton on input 001111000 would look like this:

```
Line:  1 2 3 2 3 2 4 5 2 4 5 2 4 6
Input:   0   0   1     1     1
R1:    0 0 0 1 1 2 2 2 1 1 1 0 0 0
```

Note that in a run, the values of the register per cycle always reflect those before going into the line; the updated values of the register are in the next columnl. The input reflects the symbol read in the line (if any) where "-" denotes the case that the input is exhausted.

Theorem 18.2. *Register machines can be translated into counter machines.*

Proof Idea. The main idea is that one can simulate addition, subtraction, assignment and comparison using additional registers. Here an example on how to translate the sequence $R_1 = R_2 + R_3$ into a code segment which uses an addition register R_4 which is 0 before and after the operation.

Line 1: Operation $R_1 = R_2 + R_3$ on Counter Machine
Line 2: If $R_1 = 0$ Then Goto Line 4;
Line 3: $R_1 = R_1 - 1$; Goto Line 2;
Line 4: If $R_2 = 0$ Then Goto Line 6;
Line 5: $R_4 = R_4 + 1$; $R_2 = R_2 - 1$; Goto Line 4;
Line 6: If $R_4 = 0$ Then Goto Line 8;
Line 7: $R_1 = R_1 + 1$; $R_2 = R_2 + 1$; $R_4 = R_4 - 1$; Goto Line 6;
Line 8: If $R_3 = 0$ Then Goto Line 10;
Line 9: $R_4 = R_4 + 1$; $R_3 = R_3 - 1$; Goto Line 8;
Line 10: If $R_4 = 0$ Then Goto Line 12;
Line 11: $R_1 = R_1 + 1$; $R_3 = R_3 + 1$; $R_4 = R_4 - 1$; Goto Line 10;
Line 12: Continue with Next Operation;

A further example is $R_1 = 2 - R_2$ which is realised by the following code.

Line 1: Operation $R_1 = 2 - R_2$ on Counter Machine
Line 2: If $R_1 = 0$ Then Goto Line 4;
Line 3: $R_1 = R_1 - 1$; Goto Line 2;
Line 4: $R_1 = R_1 + 1$; $R_1 = R_1 + 1$;
Line 5: If $R_2 = 0$ Then Goto Line 10;
Line 6: $R_1 = R_1 - 1$; $R_2 = R_2 - 1$;
Line 7: If $R_2 = 0$ Then Goto Line 9;
Line 8: $R_1 = R_1 - 1$;
Line 9: $R_2 = R_2 + 1$;
Line 10: Continue with Next Operation;

Similarly one can realise subtraction and comparison by code segments. Note that each time one compares or adds or subtracts a variable, the variable needs to be copied twice by decrementing and incrementing the corresponding registers, as registers compare only to 0 and the value in the register gets lost when one downcounts it to 0 so that a copy must be counted up in some other register to save the value. This register is in the above example R_4. ∎

Quiz 18.3. *Provide counter automaton translations for the following commands:*

- $R_2 = R_2 + 3$*;*
- $R_3 = R_3 - 2$*;*
- $R_1 = 2$.

Write the commands in a way that that 1 *is subtracted only from registers if those are not* 0.

Exercise 18.4. *Provide a translation for a subtraction: $R_1 = R_2 - R_3$. Here the result is 0 in the case that R_3 is greater than R_2. The values of R_2, R_3 after the translated operation should be the same as before.*

Exercise 18.5. *Provide a translation for a conditional jump: If $R_1 \leq R_2$ then Goto Line 200. The values of R_2, R_3 after doing the conditional jump should be the same as before the translation of the command.*

Corollary 18.6. *Every language recognised by a Turing machine or a register machine can also be recognised by a counter machine. In particular there are languages L recognised by counter machines for which the membership problem is undecidable.*

Theorem 18.7. *If K is recognised by a counter machine then there are deterministic context-free languages L and H and a homomorphism h such that*

$$K = h(L \cap H).$$

In particular, K is generated by some grammar.

Proof. The main idea of the proof is the following: One makes L and H to be computations such that for L the updates after an odd number of steps and for H the updates after an even number of steps is checked; furthermore, one intersects one of them, say H with a regular language in order to meet some other, easy to specify requirements on the computation.

Furthermore, $h(L \cap H)$ will consist of the input words of accepting counter machine computations; in order to achieve that this works, one requires that counter machines read the complete input before accepting. If they read only a part, this part is the accepted word, but no proper extension of it.

Now for the detailed proof, let K be the given recursively enumerable set and M be a counter machine which recognises K. Let $R_1, R_2, \ldots, R_n$ be the registers used and let $1, 2, \ldots, m$ be the line numbers used. Without loss of generality, the alphabet used is $\{0, 1\}$. One uses $0, 1$ only to denote the input symbol read in the current cycle and 2 to denote the outcome of a reading when the input is exhausted. For a line $LN \in \{1, 2, \ldots, m\}$, let 3^{LN} code the line number. Furthermore, one codes as 4^x the current value of the counter where $x = p_1^{R_1} \cdot p_2^{R_2} \cdot \ldots \cdot p_n^{R_n}$ and $p_1, p_2, \ldots, p_n$ are the first n prime numbers. For example, if $R_1 = 3$ and $R_3 = 1$ and all other registers are 0 then $x = 2^3 \cdot 3^0 \cdot 5^1 \cdot 7^0 \cdot \ldots = 40$. Thus the set I of all possible configurations is of the form

$$I = \{0, 1, 2, \varepsilon\} \cdot \{3, 33, \ldots, 3^m\} \cdot \{4\}^+$$

where the input (if requested) is the first digit then followed by the line number coded as 3^{LN} then followed by the registers coded as 4^x; note that $x > 0$ as it is the multiplication of prime powers. Furthermore, let

$$J = \{v \cdot w : v, w \in I \text{ and } w \text{ is configuration of next step after } v\}$$

be the set of all legal successor configurations. Note that J is deterministic context-free: The pushdown automaton starts with S on the stack. It has several states which permit to memorise the symbol read (if any) and the line number which is the number of 3 until the first 4 comes; if this number is below 1 or above m the pushdown automaton goes into an always rejecting state and ignores all further inputs. Then the pushdown automaton counts the number of 4 by pushing them onto the stack. It furthermore reads from the next cycle the input symbol (if any) and the new line number and then starts to compare the 4; again in the case that the format is not kept, the pushdown automaton goes into an always rejecting state and ignores all further input. Depending of the operation carried out, the pushdown automaton compares the updated memory with the old one and also checks whether the new line number is chosen adequately. Here some representative sample commands and how the deterministic pushdown automaton handles them:

Line i: $R_k = R_k + 1$;

In this case, one has that the configuration update must be of the form

$$\{3^i\} \cdot \{4\}^x \cdot \{0,1,2,\varepsilon\} \cdot \{3^{i+1}\} \cdot \{4\}^{x \cdot p_k}$$

and the deterministic pushdown automaton checks whether the new number of 3 is one larger than the old one and whether when comparing the second run of 4 those are p_k times many of the previous run, that is, it would count down the stack only after every p_k-th 4 and keep track using the state that the second number of 4 is a multiple of p_k.

Line i: $R_k = R_k - 1$;

In this case, one has that the configuration update must be of the form

$$\{3^i\} \cdot \{4\}^x \cdot \{0,1,2,\varepsilon\} \cdot \{3^{i+1}\} \cdot \{4\}^{x/p_k}$$

and the deterministic pushdown automaton checks whether the new number of 3 is one larger than the old one and whether when comparing the second run of 4 it would count down the stack by p_k symbols for each 4 read and it would use the state to check whether the first run of 4 was a multiple of p_k in order to make sure that the subtraction is allowed.

Line i: If $R_k = 0$ then Goto Line j;

In this case, the configuration update must either be of the form

$$\{3^i\} \cdot \{4\}^x \cdot \{0,1,2,\varepsilon\} \cdot \{3^j\} \cdot \{4\}^x$$

with x not being a multiple of p_k or it must be of the form

$$\{3^i\} \cdot \{4\}^x \cdot \{0,1,2,\varepsilon\} \cdot \{3^{i+1}\} \cdot \{4\}^x$$

with x being a multiple of p_k. Being a multiple of p_k can be checked by using the state and can be done in parallel with counting; the preservation of the value is done accordingly.

Line i: If input symbol is 0 then goto Line j_0; If input symbol is 1 then goto Line j_1; If input is exhausted then goto Line j_2;

Now the configuration update must be of one of the form

$$u \cdot \{3^i\} \cdot \{4\}^x \cdot \{0,1,2,\varepsilon\} \cdot \{3^{j_u}\} \cdot \{4\}^x$$

for some $u \in \{0,1,2\}$ and the deterministic pushdown automaton can use the state to memorise u, i and the stack to compare the two occurrences of 4^x. Again, if the format is not adhered to, the pushdown automaton goes into an always rejecting state and ignores all future input.

One can see that also the language J^* can be recognised by a deterministic pushdown automaton, as the automaton, after processing one word from J, has in the case of success the stack S and can now process the next word. Thus the overall language of correct computations is

$$(J^* \cdot (I \cup \{\varepsilon\})) \cap (I \cdot J^* \cdot (I \cup \{\varepsilon\})) \cap R$$

where R is a regular language which codes that the last line number is that of a line having the command ACCEPT and that the first line number is 1 and the initial value of all registers is 0 and that once a 2 is read from the input (for exhausted input) then all further attempts to read an input are answered with 2. So if the lines 5 and 8 carry the command ACCEPT then

$$R = (\{34\} \cdot I^* \cdot \{3^5, 3^8\} \cdot \{4\}^+) \cap (\{0,1,3,4\}^* \cdot \{2,3,4\}^*).$$

As the languages $J^* \cdot (I \cup \{\varepsilon\})$ and $I \cdot J^* \cdot (I \cup \{\varepsilon\})$ are deterministic context-free, one has also that $L = J^* \cdot (I \cup \{\varepsilon\})$ and $H = (I \cdot J^* \cdot (I \cup \{\varepsilon\})) \cap R$ are deterministic context-free.

Thus one can construct a context-sensitive grammar for $H \cap L$. Furthermore, let h be the homomorphism given by $h(0) = 0$, $h(1) = 1$, $h(2) = \varepsilon$, $h(3) = \varepsilon$ and $h(4) = \varepsilon$. Taking into account that in an accepting computation v accepting a word w all the input symbols are read, one then gets that $h(v) = w$. Thus $h(L \cap H)$ contains all the words accepted by the counter machine and $K = h(L \cap H)$. As $L \cap H$ are generated by a context-sensitive grammar, it follows from Proposition 4.12 that $h(L \cap H)$ is generated by some grammar. ∎

Exercise 18.8. *In the format of the proof before and with respect to the sample multi counter machine from Definition 18.1, give the encoded version (as word from* $\{0,1,2,3,4\}^+$*) of the run of the machine on the input* 001.

Exercise 18.9. *In the format of the proof before and with respect to the sample multi counter machine from Definition 18.1, give the encoded version (as word from* $\{0,1,2,3,4\}^+$*) of the run of the machine on the input* 001111000.

Theorem 18.10. *A set* $K \subseteq \Sigma^*$ *is recursively enumerable iff it is generated by some grammar. In particular, there are grammars for which it is undecidable which words they generate.*

Proof. If K is generated by some grammar, then every word w has a derivation $S \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_n$ in this grammar. It is easy to see that an algorithm can check, by all possible substitutions, whether $v_m \Rightarrow v_{m+1}$. Thus one can make a function f which on input $S \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_n$ checks whether all steps of the derivation are correct and whether $v_n \in \Sigma^*$ for the given alphabet; if these tests are passed then the function outputs v_n else the function is undefined. Thus K is the range of a partial recursive function.

The converse direction is that if K is recursively enumerable then K is recognised by a Turing machine and then K is recognised by a counter automaton and then K is generated by some grammar by the previous theorem. ∎

Corollary 18.11. *The following questions are undecidable:*

- *Given a grammar and a word, does this grammar generate the word?*
- *Given two deterministic context-free languages by deterministic push down automata, does their intersection contain a word?*
- *Given a context-free language given by a grammar, does this grammar generate* $\{0, 1, 2, 3, 4\}^*$*?*
- *Given a context-sensitive grammar, does its language contain any word?*

Proof. One uses Theorem 18.7 and one lets K be an undecidable recursively enumerable language, say a suitable encoding of the diagonal halting problem.

For the first item, if one uses a fixed grammar for K and asks whether an input word is generated by it, this is equivalent to determining the membership in the diagonal halting problem. This problem is undecidable. The problem where both, the grammar and the input word, can be varied, is even more general and thus also undecidable.

For the second item, one first produces two deterministic pushdown automata for the languages L and H. Second one considers for an input word $w = b_1 \ldots b_n \in \{0, 1\}^n$ the set

$$R_w = \{3, 4\}^* \cdot \{b_1\} \cdot \{3, 4\}^* \cdot \{b_2\} \cdot \ldots \cdot \{3, 4\}^* \cdot \{b_n\} \cdot \{2, 3, 4\}^*.$$

and notes that $L \cap H \cap R_w$ only contains accepting computations which read exactly the word w. One can construct a deterministic finite automaton for R_w and combine it with the deterministic pushdown automaton for H to get a deterministic pushdown automaton for $H_w = H \cap R_w$. Now the question whether the intersection of L and H_w is empty is equivalent to whether there is an accepting computation of the counter machine which reads the input w; this question cannot be decided. Thus the corresponding algorithm cannot exist.

For the third item, note that the complement $\{0, 1, 2, 3, 4\}^* - (L \cap H_w)$ of $L \cap H_w$ equals to $(\{0, 1, 2, 3, 4\}^* - L) \cup (\{0, 1, 2, 3, 4\}^* - H_w)$. The two parts of this union are deterministic context-free languages which have context-free grammars which can be computed from the deterministic pushdown automata for L and H_w; these two grammars can be combined to a context-free grammar for the union. Now being able to check whether this so obtained context-free grammar generates all words is equivalent to checking whether $w \notin K$ – what was impossible.

The fourth item is more or less the same as the second item; given deterministic pushdown automata for L and H_w, one can compute a context-sensitive grammar for $L \cap H_w$. Checking whether this grammar contains a word is as difficult as deciding whether $w \in K$, thus impossible. ∎

The above proof showed that it is undecidable to check whether a context-free grammar generates $\{0, 1, 2, 3, 4\}^*$. Actually this is undecidable for all alphabets with at least two symbols, so it is already undecidable to check whether a context-free grammar generates $\{0, 1\}^*$.

A further famous undecidable but recursively enumerable problem is the Post's Correspondence Problem. Once one has shown that this problem is undecidable, it provides an alternative approach to show the undecidability of the above questions in formal language theory.

Description 18.12: Post's Correspondence Problem. An instance of Post's Correspondence Problem is a list $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ of pairs of words. Such an instance has a solution iff there is a sequence $k_1, k_2, \ldots, k_m$ of numbers in $\{1, \ldots, n\}$ such that $m \geq 1$ – so that the sequence is not empty – and

$$x_{k_1} x_{k_2} \ldots x_{k_m} = y_{k_1} y_{k_2} \ldots y_{k_n},$$

that is, the concatenation of the words according to the indices provided by the sequence gives the same independently of whether one chooses the x-words or the y-words.

Consider the following pairs: (a,a), (a,amanap), (canal,nam), (man,lanac), (o,oo), (panama,a), (plan,nalp), This list has some trivial solutions like $1, 1, 1$ giving aaa for both words. It has also the famous solution $2, 4, 1, 7, 1, 3, 6$ which gives the palindrome as a solution:

```
a       man     a       plan    a       canal   panama
amanap  lanac   a       nalp    a       nam     a
```

The following instance of Post's correspondence problem does not admit any solution: (1,0), (2,135), (328,22222), (4993333434,3333), (8,999). The easiest way to see is that no pair can go first: the x-word and the y-word always start with different digits.

Exercise 18.13. *For the following version of Post's Correspondence Problem, determine whether it has a solution: (23,45), (2289,2298), (123,1258), (777,775577), (1,9999), (11111,9).*

Exercise 18.14. *For the following version of Post's Correspondence Problem, determine whether it has a solution: (1,9), (125,625), (25,125), (5,25), (625,3125), (89,8), (998,9958).*

Exercise 18.15. *One application of Post's Correspondence Problem is to get a proof for the undecidability to check whether the intersection of two deterministic context-free languages is non-empty. For this, consider an instance of Post's Correspondence Problem given by $(x_1, y_1), \ldots, (x_n, y_n)$ and assume that the alphabet Σ contains the digits $1, 2, \ldots, n, n+1$ plus all the symbols occurring in the x_m and y_m. Now let $L = \{k_m k_{m-1} \ldots k_1 (n+1) x_{k_1} x_{k_2} \ldots x_{k_m} : m > 0$ and $k_1, k_2, \ldots, k_m \in \{1, \ldots, n\}\}$ and $H = \{k_m k_{m-1} \ldots k_1 (n+1) y_{k_1} y_{k_2} \ldots y_{k_m} : m > 0$ and $k_1, k_2, \ldots, k_m \in \{1, \ldots, n\}\}$. Show that L, H are deterministic context-free and that their intersection is non-empty iff the given instance of Post's Correspondence Problem has a solution; furthermore, explain how the corresponding deterministic pushdown automata can be constructed from the instance.*

Description 18.16: Non-deterministic machines. Non-determinism can be realised in two ways: First by a not determined transition, that is, a Goto command has two different lines and the machine can choose which one to take or the Turing machine has in the table several possible successor states for some combination where it choses one. The second way to implement non-determinism is to say that a register or counter has a value x and the machine replaces x by some arbitrary value from $\{0, 1, \ldots, x\}$. In order to avoid too much computation power, the value should not go up by guessing. Non-deterministic machines can have many computations which either and in an accepting state (with some output) or in a rejecting state (where the output is irrelevant) or which never halt (when again all contents in the machine registers or tape is irrelevant). One defines the notions as follows:

- A function f computes on input x a value y iff there is an accepting run which produces the output y and every further accepting run produces the same output; rejected runs and non-terminating runs are irrelevant in this context.
- A set L is recognised by a non-deterministic machine iff for every x it holds that $x \in L$ iff there is an accepting run of the machine for this input x.

One can use non-determinism to characterise the regular and context-sensitive languages via Turing machines or register machines.

Theorem 18.17. *A language L is context-sensitive iff there is a Turing machine which recognises L and which modifies only those cells on the Turing tape which are occupied by the input iff there is a non-deterministic register machine recognising the* *language and a constant c such that the register machine on any run for an input consisting of n symbols never takes in its registers values larger than c^n.*

These machines are also called linear bounded automata as they are Turing machines whose workspace on the tape is bounded linearly in the input. One can show that a linear bound on the input and working just on the cells given as an input is not giving a different model. An open problem is whether in this characterisation the word "non-deterministic" can be replaced by "deterministic", as it can be done for finite automata.

Theorem 18.18. *A language L is regular iff there is a non-deterministic Turing machine and a linear bound $a \cdot n + b$ such that the Turing machine makes for each input consisting of n symbols in each run at most $a \cdot n + b$ steps and recognises L.*

Note that Turing machines can modify the tape on which the input is written while a deterministic finite automaton does not have this possibility. This result shows that, on a linear time constraint, this possibility does not help. This result is for Turing machines with one tape only; there are also models where Turing machines have several tapes and such Turing machines can recognise the set of palindromes in linear time though the set of palindromes is not regular. In the above characterisation, one can replace "non-deterministic Turing machine" by "deterministic Turing machine"; however, the result is stated here in the more general form.

Example 18.19. Assume that a Turing machine has as input alphabet the decimal digits $0, 1, \ldots, 9$ and as tape alphabet the additional blanc $\sqcup$. This Turing machine does the following: For an input word w, it goes four times over the word from left to right and replaces it a word v such that $w = 3v + a$ for $a \in \{0, 1, 2\}$ in decimal notation; in the case that doing this in one of the passes results in an $a \notin \{0, 1, 2\}$, the Turing machine aborts the computation and rejects. If all four passes went through without giving a non-zero remainder, the Turing machine checks whether the resulting word is of the from the set $\{0\}^* \cdot \{110\} \cdot \{0\}^* \cdot \{110\} \cdot \{0\}^*$.

One detail, left out in the overall description is how the pass divides by 3 when going from the front to the end. The method to do this is to have a memory a which is the remainder-carry and to initialise it with 0. Then, one replaces in each step the current decimal digit b by the value $(a \cdot 10 + b)/3$ where this value is down-rounded to the next integer (it is from $\{0, 1, \ldots, 9\}$) and the new value of a is the remainder of $a \cdot 10 + b$ by 3. After the replacement the Turing machine goes right.

Now one might ask what language recognised by this Turing machine is. It is the following: $\{0\}^* \cdot \{891\} \cdot \{0\}^* \cdot \{891\} \cdot \{0\}^+$. Note that 110 times 3^4 is 8910 and therefore the trailing 0 must be there. Furthermore, the nearest the two blocks of 110 can be is 110110 and that times 81 is 8918910. Thus it might be that there is no 0 between the two words 891.

Exercise 18.20. *Assume that a Turing machine does the following: It has* 5 *passes over the input word* w *and at each pass, it replaces the current word* v *by* $v/3$*. In the case that during this process of dividing by* 3 *a remainder different from* 0 *occurs for the division of the full word, then computation is aborted as rejecting. If all divisions go through and the resulting word* v *is* $w/3^5$ *then the Turing machine adds up the digits and accepts iff the sum of digits is exactly* 2 *— note that it can reject once it sees that the sum is above* 3 *and therefore this process can be done in linear time with constant memory. The resulting language is regular by Theorem 18.18. Determine a regular expression for this language.*

Exercise 18.21. *A Turing machine does two passes over a word and divides it the decimal number on the tape each time by* 7*. It then accepts iff the remainders of the two divisions sum up to* 10*, that is, either one pass has remainder* 4 *and the other has remainder* 6 *or both passes have remainder* 5*. Note that the input for the second pass is the downrounded fraction of the first pass divided by* 7*. Construct a dfa for this language.*

Exercise 18.22. *Assume that a Turing machine checks one condition, does a pass on the input word from left to right modifying it and then again checks the condition. The precise activity is the following on a word from* $\{0, 1, 2\}^*$*:*

Initialise $c = 0$ *and update* c *to* $1 - c$ *whenever a* 1 *is read (after doing the replacement). For each symbol do the following replacement and then go right:*

If $c = 0$ *then* $1 \to 0, 2 \to 1, 0 \to 0$*;*
If $c = 1$ *then* $1 \to 2, 2 \to 2, 0 \to 1$.

Here an example:

```
Before pass 0100101221010210
After pass  0011200222001220
```

The Turing machine accepts if before the pass there are an even number of 1 *and afterwards there are an odd number of* 1*.*

Explain what the language recognised by this Turing machine is and why it is regular. As a hint: interpret the numbers as natural numbers in ternary representation and analyse what the tests and the operations do.

Selftest 18.23. Provide a register machine program which computes the Fibonacci sequence. Here Fibonacci(n) $= n$ for $n < 2$ and Fibonacci(n) $=$ Fibonacci($n-1$) $+$ Fibonacci($n-2$) for $n \geq 2$. On input n, the output is Fibonacci(n).

Selftest 18.24. Define by structural induction a function F such that $F(\sigma)$ is the shortest string, if any, of the language represented by the regular expression σ. For this, assume that only union, concatenation, Kleene Plus and Kleene Star are permitted to combine languages. If σ represents the empty set then $F(\sigma) = \infty$. For example, $F(\{0011, 000111\}) = 4$ and $F(\{00, 11\}^+) = 2$.

Selftest 18.25. Construct a context-sensitive grammar for all words in $\{0\}^+$ which have length 2^n for some n.

Selftest 18.26. Construct a deterministic finite automaton recognising the language of all decimal numbers x which are multiples of 3 but which are not multiples of 10. The deterministic finite automaton should have as few states as possible.

Selftest 18.27. Determine, in dependence of the number of states of a non-deterministic finite automaton, the best possible constant which can be obtained for the following weak version of the pumping lemma: There is a constant k such that, for all words $w \in L$ with $|w| \geq k$, one can split $w = xyz$ with $y \neq \varepsilon$ and $xy^*z \subseteq L$. Prove the answer.

Selftest 18.28. Which class C of the following classes of languages is not closed under intersection: regular, context-free, context-sensitive and recursively enumerable? Provide an example of languages which are in C such that their intersection is not in C.

Selftest 18.29. Provide a homomorphism h which maps 001 and 011 to words which differ in exactly two digits and which satisfies that $h(002) = h(311)$ and $|h(23)| = |h(32)|$.

Selftest 18.30. Translate the following grammar into the normal form of linear grammars:

$$(\{S\}, \{0, 1, 2\}, \{S \to 00S11|222\}, S).$$

Furthermore, explain which additional changes one would to carry out in order to transform the linear normal form into Chomsky normal form.

Selftest 18.31. Consider the grammar

$$(\{S, T, U\}, \{0, 1\}, \{S \to ST|TT|0, T \to TU|UT|UU|1, U \to 0\}, S).$$

Use the algorithm of Cocke, Kasami and Younger to check whether 0100 is generated by this grammar and provide the corresponding table.

Selftest 18.32. Let L be deterministic context-free and H be a regular set. Which of the following sets is not guaranteed to be deterministic context-free: $L \cdot H$, $H \cdot L$, $L \cap H$ or $L \cup H$? Make the right choice and then provide examples of L, H such that the chosen set is not deterministic context-free.

Selftest 18.33. Write a register machine program which computes the function $x \mapsto x^8$. All macros used must be defined as well.

Selftest 18.34. The universal function $e, x \mapsto \varphi_e(x)$ is partial recursive. Now define ψ as $\psi(e) = \varphi_e(\mu x\,[\varphi_e(x) > 2e])$; this function is partial-recursive as one can make an algorithm which simulates $\varphi_e(0), \varphi_e(1), \ldots$ until it finds the first x such that $\varphi_e(x)$ takes a value $y > 2e$ and outputs this value y; this simulation gets stuck if one of the simulated computations does not terminate or if the corresponding input x does not exist. The range A of ψ is recursively enumerable. Prove that A is undecidable; more precisely, prove that the complement of A is not recursively enumerable.

Selftest 18.35. Let W_e be the domain of the function φ_e for an acceptable numbering $\varphi_0, \varphi_1, \ldots$ of all partial recursive functions. Construct a many-one reduction g from

$$A = \{e : W_e \text{ is infinite}\}$$

to the set

$$B = \{e : W_e = \mathbb{N}\};$$

that is, g has to be a recursive function such that W_e is infinite iff $W_{g(e)} = \mathbb{N}$.

Selftest 18.36. Is it decidable to test whether a context-free grammar generates infinitely many elements of $\{0\}^* \cdot \{1\}^*$?

Solution for Selftest 18.23. The following register program computes the Fibonacci sequence. R_2 will carry the current value and R_3, R_4 the next two values where $R_4 = R_2 + R_3$ according to the recursive equation of the Fibonacci sequence. R_5 is a counting variable which counts from 0 to R_1. When R_1 is reached, the value in R_2 is returned; until that point, in each round, R_3, R_4 are copied into R_2, R_3 and the sum $R_4 = R_2 + R_3$ is updated.

Line 1: Function Fibonacci(R_1);
Line 2: $R_2 = 0$;
Line 3: $R_3 = 1$;
Line 4: $R_5 = 0$;
Line 5: $R_4 = R_2 + R_3$;
Line 6: If $R_5 = R_1$ Then Goto Line 11;
Line 7: $R_2 = R_3$;
Line 8: $R_3 = R_4$;
Line 9: $R_5 = R_5 + 1$;
Line 10: Goto Line 5;
Line 11: Return(R_2).

Solution for Selftest 18.24. One can define F as follows. For the base cases, F is defined as follows:

- $F(\emptyset) = \infty$;
- $F(\{w_1, w_2, \ldots, w_n\}) = \min\{|w_m| : m \in \{1, \ldots, n\}\}$.

In the inductive case, when $F(\sigma)$ and $F(\tau)$ are already known, one defined $F(\sigma \cup \tau)$, $F(\sigma \cdot \tau)$, $F(\sigma^*)$ and $F(\sigma^+)$ as follows:

- If $F(\sigma) = \infty$ then $F(\sigma \cup \tau) = F(\tau)$;
  If $F(\tau) = \infty$ then $F(\sigma \cup \tau) = F(\sigma)$;
  If $F(\sigma) < \infty$ and $F(\tau) < \infty$ then $F(\sigma \cup \tau) = \min\{F(\sigma), F(\tau)\}$;
- If $F(\sigma) = \infty$ or $F(\tau) = \infty$
  then $F(\sigma \cdot \tau) = \infty$
  else $F(\sigma \cdot \tau) = F(\sigma) + F(\tau)$;
- $F(\sigma^*) = 0$;
- $F(\sigma^+) = F(\sigma)$.

Solution for Selftest 18.25. The grammar contains the non-terminals S, T, U and the terminal 0 and the start symbol S and the following rules: $S \to 0|00|T0U$, $T \to$

TV, $V0 \to 00V$, $VU \to 0U$, $T \to W$, $W0 \to 00W$, $WU \to 00$. Now $S \Rightarrow T0U \Rightarrow W0U \Rightarrow 00WU \Rightarrow 0000$ generates 0^4. Furthermore, one can show by induction on n that $S \Rightarrow^* T0^{2^n-1}U \Rightarrow TV0^{2^n-1}U \Rightarrow^* T0^{2^{n+1}-2}VU \Rightarrow T0^{2^{n+1}-1}U$ and $S \Rightarrow^* T0^{2^n-1}U \Rightarrow W0^{2^n-1}U \Rightarrow^* 0^{2^{n+1}-2}WU \Rightarrow 0^{2^{n+1}}$. So, for each n, one can derive $0^{2^{n+1}}$ and one can also derive $0, 00$ so that all words from $\{0\}^+$ of length 2^n can be derived.

Solution for Selftest 18.26. The deterministic finite automaton needs to memorise two facts: the remainder by three and whether the last digit was a 0; the latter needs only to be remembered in the case that the number is a multiple of 3. So the dfa has four states: s, q_0, q_1, q_2 where s is the starting state and q_0, q_1, q_2 are the states which store the remainder by 3 of the sum of the digits seen so far. The transition from state s or q_a ($a \in \{0, 1, 2\}$) on input $b \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is as follows (where also $a = 0$ in the case that the state is s):

- If $a + b$ is a multiple of 3 and $b = 0$ then the next state is s;
- If $a + b$ is a multiple of 3 and $b \neq 0$ then the next state is q_0;
- If $a + b$ has the remainder $c \in \{1, 2\}$ modulo 3 then the next state is q_c.

Furthermore, s is the start state and q_0 is the only accepting state.

Solution for Selftest 18.27. Assume that L is recognised by a non-deterministic finite automaton having n states. Then the following holds: For every word $w \in L$ of length n or more, one can split $w = xyz$ such that $y \neq \varepsilon$ and $xy^*z \subseteq L$. For this one considers an accepting run of the nfa on the word w which is a sequence $q_0 q_1 \ldots q_n$ of states where q_m is the state after having processed m symbols, so q_0 is the initial state. The state q_n must be accepting. As there are $n + 1$ values $q_0, q_1, \ldots, q_n$ but only n states in the automaton, there are i, j with $0 \leq i < j \leq n$ such that $q_i = q_j$. Now let x be the first i symbols of w, y be the next $j - i$ symbols and z be the last $n - j$ symbols of w, clearly $w = xyz$ and $|y| = j - i > 0$. It is easy to see that when y is omitted then $q_0 q_1 \ldots q_i q_{j+1} \ldots q_n$ is a run of the automaton on xz and if y is repeated, one can repeat the sequence from $q_i to q_j$ accordingly. So $q_0 \ldots q_i (q_{i+1} \ldots q_j)^3 q_{j+1} \ldots q_n$ is an accepting run on xy^3z. Thus all words in xy^*z are accepted by the non-deterministic finite automaton and $xy^*z \subseteq L$.

Furthermore, there are for each n finite automata with n states which accept all words having at most $n - 1$ symbols, they advance from one state to the next upon reading a symbol and get stuck once all states are used up. Thus the pumping constant cannot be $n - 1$, as otherwise the corresponding language would need to have infinitely many words, as a word of length $n - 1$ could be pumped. So n is the optimal constant.

Solution for Selftest 18.28. The context-free languages are not closed under intersection. The example is the language $\{0^n1^n2^n : n \in \mathbb{N}\}$ which is the intersection of the two context-free languages $\{0^n1^n2^m : n,m \in \mathbb{N}\}$ and $\{0^n1^m2^m : n,m \in \mathbb{N}\}$. Both languages are context-free; actually they are even linear languages.

Solution to Selftest 18.29. One can choose the homomorphism given by $h(0) = 55$, $h(1) = 66$, $h(2) = 6666$ and $h(3) = 5555$. Now $h(001) = 555566$ and $h(011) = 556666$ so that they differ in two positions and $h(002) = h(311) = 55556666$. Furthermore, $|h(23)| = |h(32)|$ is true for every homomorphism and a vacuous condition.

Solution to Selftest 18.30. The grammar can be translated into the normal form for linear grammars as follows: The non-terminals are $S, S', S'', S''', S''', T, T'$ and the rules are $S \to 0S'|2T$, $S' \to 0S''$, $S'' \to S'''1$, $S''' \to S1$, $T \to 2T'$, $T' \to 2$.

For Chomsky Normal form one would have to introduce two further non-terminals V, W representing 0 and 1 and use that $T' \to 2$. Then one modifies the grammar such that the terminals do not appear in any right side with two non-terminals. The updated rules are the following: $S \to VS'|T'T$, $S' \to VS''$, $S'' \to S'''W$, $S''' \to SW$, $T \to T'T'$, $T' \to 2$, $V \to 0$, $W \to 1$.

Solution for Selftest 18.31. The given grammar is $(\{S,T,U\}, \{0,1\}, \{S \to ST|TT|$ $0, T \to TU|UT|UU|1, U \to 0\}, S)$. Now the table for the word 0100 is the following:

$$\begin{array}{ccccccc}
 & & & E_{1,4} = \{S,T\} & & & \\
 & & E_{1,3} = \{S,T\} & & E_{2,4} = \{S,T\} & & \\
 & E_{1,2} = \{S,T\} & & E_{2,3} = \{T\} & & E_{3,4} = \{T\} & \\
E_{1,1} = \{S,U\} & & E_{2,2} = \{T\} & & E_{3,3} = \{S,U\} & & E_{4,4} = \{S,U\} \\
0 & & 1 & & 0 & & 0
\end{array}$$

As $S \in E_{1,4}$, the word 0100 is in the language.

Solution for Selftest 18.32. If L is deterministic context-free and H is regular then $L \cap H$, $L \cup H$ and $L \cdot H$ are deterministic context-free. However, the set $H \cdot L$ might not be deterministic context-free. An example is the following set: $H = (\{0\}^* \cdot \{1\}) \cup \{\varepsilon\}$ and $L = \{0^n10^n : n \in \mathbb{N}\}$. L is one of the standard examples of deterministic context-free sets; however, when a deterministic pushdown automaton processes an input starting with 0^n10^n, it has to check whether the number of 0 before the 1 and after the 1 are the same and therefore it will erase from the stack the information on how many 0 are there. This is the right thing to do in the case that the input is from $\{\varepsilon\} \cdot L$. However, in the case that the input is from $\{0\}^* \cdot \{1\} \cdot L$, the deterministic pushdown automaton has now to process in total an input of the form $0^n10^n10^m$ which will be accepted iff $n = m$. The information on what n was is, however, no longer available.

Solution for Selftest 18.33. One first defines the function Square computing $x \mapsto x^2$.

Line 1: Function Square(R_1);
Line 2: $R_3 = 0$;
Line 3: $R_2 = 0$;
Line 4: $R_2 = R_2 + R_1$;
Line 5: $R_3 = R_3 + 1$;
Line 6: If $R_3 < R_1$ then goto Line 4;
Line 7: Return(R_1).

Now one defines the function $x \mapsto x^8$.

Line 1: Function Eightspower(R_1);
Line 2: $R_2 =$ Square(R_1);
Line 3: $R_3 =$ Square(R_2);
Line 4: $R_4 =$ Square(R_3);
Line 5: Return(R_4).

Solution for Selftest 18.34. First note that the complement of A is infinite: All elements of $A \cap \{0, 1, \ldots, 2e\}$ must be from the finite set $\{\psi(0), \psi(1), \ldots, \psi(e-1)\}$ which has at most e elements, thus there must be at least e non-elements of A below $2e$. If the complement of A would be recursively enumerable then $\mathbb{N} - A$ is the range of a function φ_e which is defined for all x. Thus $\psi(e)$ would be $\varphi_e(x)$ for the first x where $\varphi_e(x) > 2e$. As the complement of A is infinite, this x must exist. But then $\psi(e)$ is in both: it is in A by the definition of A as range of ψ and it is in the range of φ_e which is the complement of A. This contradiction shows that the complement of A cannot be the range of a recursive function and therefore A cannot be recursive.

Solution for Selftest 18.35. The task is to construct a many-one reduction g from $A = \{e : W_e \text{ is infinite}\}$ to the set $B = \{e : W_e = \mathbb{N}\}$.

For this task one first defines a partial recursive function f as follows: Let M be a universal register machine which simulates on inputs e, x the function $\varphi_e(x)$ and outputs the result iff that function terminates with a result; if the simulation does not terminate then M runs forever. Now let $f(e, x)$ is the first number t (found by exhaustive search) such that there are at least x numbers $y \in \{0, 1, \ldots, t\}$ for which $M(e, y)$ terminates within t computation steps. Note that $f(e, x)$ is defined iff W_e has at least x elements. There is now a recursive function g such that $\varphi_{g(e)}(x) = f(e, x)$ for all e, x where either both sides are defined and equal or both sides are undefined. If the domain W_e of φ_e is infinite then $\varphi_{g(e)}$ is defined for all x and $W_{g(e)} = \mathbb{N}$; if the domain W_e of φ_e has exactly y elements then $f(e,x)$ is undefined for all $x > y$ and $W_{g(e)}$ is a finite set. Thus g is a many-one reduction from A to B.

Solution for Selftest 18.36. It is decidable: The way to prove it is to construct from the given context-free grammar for some set L a new grammar for the intersection $L \cap \{0\}^* \cdot \{1\}^*$, then to convert this grammar into Chomsky Normal form and then to run the algorithm which checks whether this new grammar generates an infinite set.

19 Regular Languages and Learning Theory

Angluin [3] investigated the question on how to learn a dfa by a dialogue between a learner (pupil) and teacher. The learner can ask questions to the teacher about the concept (dfa) to be learnt and the teacher answers. The learner can ask two types of questions:

- Is the following dfa equivalent to the one to be learnt?
- Does the dfa to be learnt accept or reject the following word w?

The first type of questions are called "equivalence queries" and the second type of questions are called "membership queries". The teacher answers an equivalence query either with "YES" (then the learner has reached the goal) or "NO" plus a counterexample w on which the dfa given by the learner and the dfa to be learnt have different behaviour; the teacher answers a membership query by either "YES" or "NO".

Theoretically, the learner could just take a listing of all dfas and ask "Is dfa$_1$ correct?", "Is dfa$_2$ correct?", "Is dfa$_3$ correct?" ... and would need s equivalence queries to find out whether dfa$_s$ is correct. This strategy is, however, very slow; as there are more than 2^n dfas with n states, one would for some dfas with n states need more than 2^n queries until the dfa is learnt. Angluin showed that there is a much better algorithm and she obtained the following result.

Theorem 19.1: Angluin's algorithm to learn dfas by queries [3]. *There is a learning algorithm which has polynomial response time in each step and which learns in time polynomial in the number of states of the dfa to be learnt and the longest counterexample given an arbitrary dfa using equivalence queries and membership queries.*

Proof. A simplified version of Angluin's algorithm is given. The idea of Angluin is that the learner maintains a table (S, E, T) which is updated in each round. In this table, S is a set of words which represent the set of states. E consists of all the counterexamples observed plus their suffixes. S and E are finite sets of size polynomial in the number and length of counterexamples seen so far and T is a function which for all members $w \in S \cdot E \cup S \cdot \Sigma \cdot E$ says whether the automaton to be learnt accepts or rejects w.

Angluin defines the notion of a row: For $u \in S \cup S \cdot \Sigma$, let $(v_1, v_2, \ldots, v_k)$ be a listing of the current elements in E and for each $u \in S \cup S \cdot \Sigma$, let the vector $row(u)$ be $(T(uv_1), T(uv_2), \ldots, T(uv_k))$. The table (S, E, T) is called closed, if for every $u \in S$ and $a \in \Sigma$ there is a $u' \in S$ with $row(u') = row(ua)$.

Now Anlguin defines for each closed (S, E, T) the finite automaton DFA(S, E, T) where the set of states is S, the alphabet is Σ and the transition function finds to a state u and $a \in \Sigma$ the unique $u' \in S$ with $row(u') = row(ua)$. The starting state is represented by the empty word ε (which is in S). A state u is accepting iff $T(u) = 1$. Note that DFA(S, E, T) is complete and deterministic. The learning algorithm is now the following.

Teacher has regular set L and learner makes membership and equivalence queries.

1. Initialise $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$.
2. For all $w \in S \cdot E \cup S \cdot \Sigma \cdot E$ where $T(w)$ is not yet defined, make a membership query to determine $L(w)$ and let $T(w) = L(w)$.
3. If there are $u \in S$ and $a \in \Sigma$ with $row(ua) \neq row(u')$ for all $u' \in S$ then let $S = S \cup \{ua\}$ and go to 2.
4. Make an equivalence query whether DFA(S, E, T) recognises L.
5. If the answer is "YES" then terminate with DFA(S, E, T).
6. If the answer is "NO" with counterexample w then let $E = E \cup \{v : \exists u\, [uv = w]\}$ and go to 2.

Now one shows various properties in order to verify the termination of the algorithm and the polynomial bounds on the number of membership and equivalence queries. For this, assume that $(Q, \Sigma, \delta, s, F)$ is the minimal dfa recognising L. Now various invariants are shown.

If u, u' are different elements of S then $\delta(s, u) \neq \delta(s, u')$ as there is a word $v \in E$ with $L(uv) \neq L(u'v)$. Hence $|S| \leq |Q|$ throughout the algorithm.

If (S, E, T) is closed and $w \in E$ then the DFA accepts w iff $w \in L$.

To see this, one does the following induction: Let $w = a_1 a_2 \ldots a_n$. Clearly $T(w) = L(w)$ by the corresponding membership query. For $m = 0, 1, \ldots, n$, one shows that the automaton is after processing $a_1 a_2 \ldots a_m$ is in a state u_m with $T(u_m a_{m+1} a_{m+2} \ldots a_n) = T(w)$. This is true for $m = 0$ as $u_0 = \varepsilon$ is the initial state of DFA(S, E, T). Assume now that it is correct for $m < n$. Then u_{m+1} is the unique state in S with $row(u_{m+1}) = row(u_m a_{m+1})$. It follows that $T(u_m a_{m+1} v) = T(u_{m+1} v)$ for $v = a_{m+2} a_{m+3} \ldots a_n$. Hence the induction hypothesis is preserved and DFA(S, E, T) is after processing the full word in a state u_n with $T(u_n) = T(w)$. This state is accepting iff $T(u_n) = 1$ iff $T(w) = 1$. Hence DFA(S, E, T) is correct on w.

Assume that the algorithm has the parameters (S, E, T) before observing counterexample w and has the parameters (S', E', T') after it has done all the updates before the next equivalence query is made. Then $S \subset S'$.

Let $row_E(u)$ and $row_{E'}(u)$ denote the rows of u based on E and E'; note that $E \subseteq E'$ and therefore $row_E(u) \neq row_E(u') \Rightarrow row_{E'}(u) \neq row_{E'}(u')$. Now, as

DFA$(S,E,T) \neq$ DFA(S',E',T') on w, the states in which these two automata are after processing w must be different. As both dfas have the initial state ε, there must be a first prefix of the form ua of w such that the two automata are in different states u', u'' after processing ua. Now $row_E(ua) = row_E(u')$ and $u' \in S$ and $row_{E'}(ua) = row_{E'}(u'')$. It cannot be that $u'' \in S - \{u'\}$, as then $row_E(u'') \neq row_E(ua)$. Hence u'' must be a new state in $S' - S$ and $S \subset S'$.

Let r be the sum of all lengths of the counterexamples observed. The algorithm makes at most $|Q|$ equivalence queries and at most $|Q| \cdot (|\Sigma|+1) \cdot (r+1)$ membership queries.

As seen, $|S| \leq |Q|$ throughout the algorithm. As each equivalence query increases the size of S, there are at most $|Q|$ equivalence queries. Furthermore, E contains all non-empty prefixes of counterexamples observed plus ε, hence $|E| \leq r+1$. Now the table T has at each time the domain $S \cdot E \cup S \cdot \Sigma \cdot E$ what gives then the bound on the number of membership queries.

The overall runtime of each update is polynomial in the size of the counterexamples observed so far and in $|Q|$. So latest when $|S| = |Q|$ the answer to the equivalence query is "YES" and the learner has learnt the language L. ▮

Remark 19.2: Angluin's original algorithm [3]**.** Angluin did not put the suffixes of the counterexamples into E but she put the prefixes of the counterexamples into S. Therefore, S could contain words u, u' with $row(u) = row(u')$. In order to avoid that this is harmful, Angluin increased then E so long until the table T is consistent, that is, if $row(u) = row(u')$ then $row(ua) = row(u'a)$ for all $u, u' \in S$ and $a \in \Sigma$. This consistency requirement was explicitly added into the algorithm. The verification of the original algorithm is given in Angluin's paper [3].

Teacher has regular set L and learner makes membership and equivalence queries.

1. Initialise $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$.
2. For all $w \in S \cdot E \cup S \cdot \Sigma \cdot E$ where $T(w)$ is not yet defined, make a membership query to determine $L(w)$ and let $T(w) = L(w)$.
3. If there are $u, u' \in S$, $a \in \Sigma$ and $v \in E$ such that $row(u) = row(u')$ and $T(uav) \neq T(u'av)$ then let $E = E \cup \{av\}$ and go to 2.
4. If there are $u \in S$ and $a \in \Sigma$ with $row(ua) \neq row(u')$ for all $u' \in S$ then let $S = S \cup \{ua\}$ and go to 2.
5. Make an equivalence query whether DFA(S,E,T) recognises L.
6. If the answer is "YES" then terminate with DFA(S,E,T).
7. If the answer is "NO" with counterexample w then let $S = S \cup \{u : \exists v\,[uv = w]\}$ and go to 2.

Description 19.3: Learning from positive data [2, 4, 31]**.** Gold [31] introduced a general framework of learning in the limit. His idea was that a learner reads more and more data and at the same time outputs conjectures; from some time on, the learner should always output the same correct conjecture. More precisely, the learner consists of a memory mem and an update function uf. In each round, the update function uf maps pairs (mem, x) consisting of the current memory and a current datum x observed to pairs (mem', e) where mem' is the new memory which is based on some calculations and intended to have incorporated some way to memorise x and where e is the conjectured hypothesis. In the case of learning regular languages, this hypothesis could just be a dfa. Gold [31] observed already in his initial paper that a class is unlearnable iff it contains an infinite set and all of its finite sets. As Σ^* and each of its finite subsets is regular, the class of regular sets is not learnable from positive data. Nevertheless, one still might learn some subclasses of regular languages.

For this, one considers so called automatic families. An automatic family is given by an index set I and a family of sets $\{L_d : d \in I\}$ such that the relation of all (d, x) with $x \in L_d$ is automatic, that is, the set $\{conv(d, x) : d \in I \wedge x \in L_d\}$ is a regular set.

Here the size of the minimal index of each language is invariant up to a constant with respect to different indexings. So given two indexed families $\{L_d : d \in I\}$ and $\{H_e : e \in J\}$, one can define the automatic functions $i : I \to J$ and $j : J \to I$ with $i(d) = \min_{ll}\{e \in J : H_e = L_d\}$ and $j(e) = \min_{ll}\{d \in I : L_d = H_e\}$. Then there is a constant k such that the following holds: if $i(d)$ is defined then $|i(d)| \leq |d| + k$; if $j(e)$ is defined then $|j(e)| \leq |e| + k$. Hence the sizes of the minimal indices of a language in both families differ at most by k [40].

The data on the language L to be learnt are presented in form of a text. A text T for a language L is an infinite sequence of words and pause symbols $\#$ such that $L = \{w : w \neq \# \wedge \exists n\, [T(n) = w]\}$. The learner starts now with some fixed initial memory mem_0 and initial hypothesis e_0, say ε and a hypothesis for the empty set. In round n, the new memory and hypothesis are computed by the update function uf: $(mem_{n+1}, e_{n+1}) = uf(mem_n, T(n))$. The learner learns L using the hypothesis space $\{L_d : d \in I\}$ iff there is a $d \in I$ with $L_d = L$ and $\forall^\infty n\, [e_n = d]$.

Angluin [2] showed in a very general framework a learnability result which covers the case of automatic families.

Theorem 19.4: Angluin's tell-tale criterion [2]. *An automatic family $\{L_d : d \in I\}$ is learnable from positive data iff there is for every $d \in I$ a finite subset $F_d \subseteq L_d$ such that there is no further index with $F_d \subseteq L_e \subset L_d$.*

Proof. Assume that $\{L_d : d \in I\}$ has a learner. Blum and Blum [4] showed that for each L_d there must be a finite initial part $T(0)T(1)\ldots T(n)$ of a text for L_d

such that for every extension $T(n+1)T(n+2)\ldots T(m)$ using elements from L_d and pause symbols # it holds that the learner conjectures an index for L_d after processing $T(0)T(1)\ldots T(m)$. If such an initial part would not exist, one could inductively define a text T for L_d on which the learner infinitely often outputs an index for a set different from L_d. Now $F_d = \{T(m) : m \leq n \wedge T(m) \neq \#\}$. This is obviously a subset of L_d; furthermore, when seeing only data of L_d after this initial part $T(0)T(1)\ldots T(n)$, the learner outputs a conjecture for L_d, hence the learner does not learn any proper subset of L_d from a text starting with $T(0)T(1)\ldots T(n)$. It follows that there cannot be any e with $F_d \subseteq L_e \subset L_d$.

For the other direction, consider that the sets F_d exist. Therefore the following value $f(d)$ is defined for every $d \in I$:

$$f(d) = \min{}_{ll}\{b : \forall e \in I \;\; [\{x \in L_d : x \leq_{ll} b\} \subseteq L_e \subseteq L_d \Rightarrow L_e = L_d]\}.$$

Then it is clear that one can choose $F_d = \{x \in L_d : x \leq_{ll} f(b)\}$. One can first-order define the subset-relation and equality-relation on sets:

$$\begin{aligned} L_d \subseteq L_e \quad &\Leftrightarrow \quad \forall x\,[x \in L_d \Rightarrow x \in L_e]; \\ L_d = L_e \quad &\Leftrightarrow \quad \forall x\,[x \in L_d \Leftrightarrow x \in L_e]. \end{aligned}$$

Hence the function f is first-order definable using automatic parameters and is automatic. Thus one can make the following learning algorithm which for doing its search archives all the data seen so far: mem_n is a list of data seen so far; e_n is the least member of I which satisfies that all elements of L_d up to $f(d)$ have been observed so far and no non-elements of L_d have been observed prior to round n; if such an index does currently not exist, the learner can output ? in order to signal that there is no valid hypothesis.

Assume that the learner reads a text for a language and that d is the minimal index of this language. Assume that n is so large that the following condition are satisfied:

- For every $w \in L_d$ with $w \leq_{ll} f(d)$ there is an $m < n$ with $T(m) = w$;
- For every $e <_{ll} d$ with $L_e \not\supseteq L_d$ there is an $m < n$ with $T(m) \in L_d - L_e$.

Note that if $e <_{ll} d$ and $L_e \supset L_d$ then there must be an element $w \in L_e - L_d$ with $w \leq_{ll} f(e)$; this element does not appear in the text T. Hence, for the n considered above it holds that the hypothesis of the learner is d. Thus the learner converges on the text T to the minimal index d of the language described by the text T; it follows that the learner learns the family $\{L_d : d \in I\}$ from positive data. ∎

This characterisation answers when a class is learnable in general. One could now ask what additional qualities could be enforced on the learner for various classes. In particular, can one make the update function uf automatic? Automatic learners are defined as follows.

Description 19.5: Automatic learners [11, 39, 40]**.** An automatic learner is given by its initial memory mem_0, initial hypothesis e_0 and the update function uf which computes in round n from $conv(mem_n, x_n)$ the new memory and the hypothesis, represented as $conv(mem_{n+1}, e_{n+1})$. An automatic learner for an indexed family $\{L_d : d \in I\}$ (which is assumed to be one-one) might use another hypothesis space $\{H_e : e \in J\}$ and must satisfy that there is an n with $H_{e_n} = L_d$ and $e_m \in \{e_n, ?\}$ for all $m > n$ where ? is a special symbol the learner may output if memory constraints do not permit the learner to remember the hypothesis.

Memory constraints are there to quantify the amount of information which an automatic learner is permitted to archive on data seen in the past. In general, this data never permits to recover the full sequence of data observed, although it is in many cases still helpful. The following memory constraints can be used while learning L_d where the current conjecture of the learner is $H_{e_{n+1}}$ and where $x_0, x_1, \ldots, x_n$ are the data observed so far; i is the function with $L_{i(e)} = H_e$ for all e representing a language in the class to be learnt.

None: The automaticity of uf gives that even in the absence of an explicit constraint it holds that $|mem_{n+1}| \leq \max\{|x_n|, |mem_n|\} + k$ for some constant k and all possible values of mem_n and x_n.

Word-sized: $|mem_{n+1}| \leq \max\{|x_0|, |x_1|, \ldots, |x_n|\} + k$ for some constant k.

Hypothesis-sized: $|mem_{n+1}| \leq |e_{n+1}| + k$ for some constant k.

Original-hypothesis-sized: $|mem_{n+1}| \leq |i(e_{n+1})| + k$ for some constant k with the additional constraint that $i(e_{n+1})$ is defined, that is, $H_{e_{n+1}}$ must be in the class to be learnt.

Target-sized: $|mem_{n+1}| \leq |d| + k$ for some constant k.

Constant: $mem_{n+1} \in C$ for a fixed finite set C of possible memory values.

Memoryless: $mem_{n+1} = mem_0$.

Note that target-sized always refers to the size of the original target; otherwise the constraint would not be the same as hypothesis-sized, as the learner could use a hypothesis space where every language has infinitely many indices and would choose at every revision a hypothesis longer than the current memory size. Note that one could fix the constant k to 1 for word-sized, hypothesis-sized, original-hypothesis-sized and target-sized learners as one can adjust the alphabet-size and store the last k symbols in a convolution of one symbol. As this would, however, make the construction of learners at various times more complicated, it is easier to keep the constant k unspecified.

Furthermore, a learner is called iterative iff $mem_n = e_n$ for all n and e_0 is a hypothesis for the empty set (which is added to the hypothesis space, if needed). Iterative learners automatically have a hypothesis-sized memory; furthermore, one writes $uf(e_n, x_n) = e_{n+1}$ in place of $uf(e_n, x_n) = conv(e_{n+1}, e_{n+1})$ in order to simplify the notation.

Example 19.6. If I is finite then there is a bound b such that for all different d, d' there is an $w \leq_{ll} b$ which is in one but not both of $L_d, L_{d'}$. Hence one can make a learner which memorises for every $w \leq_{ll} b$ whether the datum w has been observed. In the limit, the learner knows for every $w \leq_{ll} b$ whether $w \in L_d$ for the language L_d to be learnt and therefore the learner will eventually converge to the right hypothesis. The given learner has constant-sized memory.

If one would require that the learner repeats the correct conjecture forever once it has converged to the right index, then only finite classes can be learnt with constant-sized memory. If one permits ? after convergence, then a memoryless learner can learn the class of all $L_d = \{d\}$ with $d \in I$ for any given infinite regular I: the learner outputs d on datum d and ? on datum # and does not keep any records on the past.

Example 19.7. Assume that $\Sigma = \{0, 1, 2\}$ and that $I = \{conv(v, w) : v, w \in \Sigma^* \wedge v \leq_{lex} w\} \cup \{conv(3, 3)\}$ with $L_{conv(v,w)} = \{u \in \Sigma^* : v \leq_{lex} u \leq_{lex} w\}$ for all $conv(v, w) \in I$. Note that $L_{conv(3,3)} = \emptyset$.

This class has an iterative learner whose initial memory is $conv(3, 3)$. Once it sees a word $u \in \Sigma^*$, the learner updates to $conv(u, u)$. From that onwards, the learner updates the memory $conv(v, w)$ on any word $u \in \Sigma^*$ to $conv(\min_{lex}\{u, v\}, \max_{lex}\{u, w\})$. This hypothesis always consists of the convolution of the lexicographically least and greatest datum seen so far and the sequence of hypotheses has converged once the learner has seen the lexicographically least and greatest elements of the set to be learnt (which exist in all languages in the class to be learnt).

Data seen so far	Hypothesis	Language of hypothesis
—	conv(3,3)	$\emptyset$
#	conv(3,3)	$\emptyset$
# 00	conv(00,00)	$\{00\}$
# 00 0000	conv(00,0000)	$\{00, 000, 0000\}$
# 00 0000 1	conv(00,1)	$\{u : 00 \leq_{lex} u \leq_{lex} 1\}$
# 00 0000 1 0	conv(0,1)	$\{u : 0 \leq_{lex} u \leq_{lex} 1\}$
# 00 0000 1 0 112	conv(0,112)	$\{u : 0 \leq_{lex} u \leq_{lex} 112\}$
# 00 0000 1 0 112 011	conv(0,112)	$\{u : 0 \leq_{lex} u \leq_{lex} 112\}$

Exercise 19.8. *Make an automatic learner which learns the class of all $L_d = \{dw : w \in \Sigma^*\}$ with $d \in \Sigma^*$; that is, $I = \Sigma^*$ in this case.*

Exercise 19.9. *Assume that a class $\{L_d : d \in I\}$ is given with $L_d \neq L_{d'}$ whenever $d, d' \in I$ are different. Assume that an automatic learner uses this class as a hypothesis space for learning satisfying any of the constraints given in Description 19.5. Let $\{H_e : e \in J\}$ be any other automatic family containing $\{L_d : d \in I\}$ as a subclass. Show that there is an automatic learner satisfying the same type of memory constraints conjecturing indices taken from J in place of I.*

Theorem 19.10: Jain, Luo and Stephan [39]. *Let $I = \Sigma^*$, $L_\varepsilon = \Sigma^+$ and $L_d = \{w \in \Sigma^* : w <_{ll} d\}$ for $d \in \Sigma^+$. The class $\{L_d : d \in I\}$ can be learnt using a word-sized memory but not using an hypothesis-sized memory.*

Proof. First assume by way of contradiction that a learner could learn the class using some chosen hypothesis space with hypothesis-sized memory. Let $T(n)$ be the n-th string of Σ^+. When learning from this text, the learner satisfies $e_m = e_{m+1}$ for some n and all $m \geq n$; furthermore, $H_{e_m} = \Sigma^+$ for these $m \geq n$. Therefore, from n onwards, all values of the memory are finite strings of length up to $|e_n| + k$ for some constant k. There are only finitely many such strings and therefore there must be $m, k \geq n$ with $mem_m = mem_k$. If one now would change the text to $T(h) = \varepsilon$ for all $h \geq m$ or $h \geq k$, respectively, the learner would converge to the same hypothesis on both of these texts, although it would be a text for either the first $m + 1$ or the first $k + 1$ strings in Σ^*. Thus the learner fails to learn at least one of these finite sets and cannot learn the class.

Second consider a word-sized learner. This learner memorises the convolution of the length-lexicographically least and greatest words seen so far. There are three cases:

- In the case that no word has been seen so far, the learner outputs ? in order to abstain from a conjecture;
- In the case that these words are ε and v, the learner conjectures $L_{Succ_{ll}(v)} = \{w : \varepsilon \leq_{ll} w \leq_{ll} v\}$;
- In the case that the words u and v memorised are different from ε, the learner conjectures $L_\varepsilon = \Sigma^+$.

The memory of the learner is either a special symbol for denoting that no word (except #) has been seen so far or the convolution of two words observed whose length is bounded by the length of the longest word seen so far. Hence the memory bound of the learner is satisfied. ∎

Theorem 19.11. *Assume that* $\Sigma = \{0,1\}$ *and* $I = \{0,1\}^* \cup \{2,3\} \cup \{conv(v,w) : v,w \in \{0,1\}^* \wedge v <_{ll} w\}$ *where the convolution is defined such that this unions are disjoint. Furthermore, let* $L_2 = \emptyset$, $L_3 = \Sigma^*$, $L_v = \{v\}$ *for* $v \in \Sigma^*$ *and* $L_{conv(v,w)} = \{v,w\}$ *for* $v,w \in \Sigma^*$ *with* $v <_{ll} w$. *The class* $\{L_d : d \in I\}$ *can neither be learnt with constant memory nor with target-sized memory. It can, however, be learnt using an original-hypothesis-sized memory.*

Proof. Assume that some learner with constant-sized memory learns this class. There is a constant k so large that (1) $|e_n| \leq |x_n|+k$ on datum x_n and at memory $mem_n \in C$ and (2) $|\max(H_e)| \leq |e|+k$ whenever H_e is finite. As C has only finitely many values, this constant k must exist. Now assume that v is any member of Σ^* and $w \in \Sigma^*$ is such that $|w| > |v|+2k+1$. Then, whenever the hypothesis e_{m+1} is computed from e_m and either v or #, the set $H_{e_{m+1}}$ is neither $\{w\}$ nor $\{v,w\}$. Hence, when the learner sees w as the first datum, it must conjecture $\{w\}$ as all subsequent data might by # and $\{w\}$ cannot be conjectured again. Furthermore, if the learner subsequently sees only v, then it cannot conjecture $\{v,w\}$. Hence, either the learner does not learn $\{w\}$ from the text $w, \#, \#, \ldots$ or the learner does not learn $\{v,w\}$ from the text $w, v, v, \ldots$; thus the learner does not learn the given class.

As Σ^* is in the class to be learnt and every data observed is consistent with the possibility that Σ^* is the language observed, every target-sized learner has at every moment to keep the index shorter than the index of Σ^* plus some constant, hence this learner has actually to use constant-sized memory what is impossible by the previous paragraph.

So it remains to show that one can learn the class by hypothesis-sized memory. This is done by showing that the class has actually an iterative learner using I as hypothesis space. Hence every hypothesis is from the original space and so the learner's memory is original-hypothesis-sized. Initially, the learner conjectures 2 until it sees a datum $v \neq \#$. Then it changes to conjecturing H_v until it sees a datum $w \notin \{\#, v\}$. Then the learner updates to $H_{conv(\min_{ll}\{v,w\},\max_{ll}\{v,w\})}$. The learner keeps this hypothesis until it sees a datum outside $\{\#, v, w\}$; in that case it makes a last mind change to $H_3 = \Sigma^*$. It is easy to see that the learner is iterative and needs only the current hypothesis as memory; furthermore, the learner is also easily seen to be correct. ▮

Theorem 19.12. *If a learner learns a class with target-sized memory then the learner's memory is also word-sized on texts for languages in the class.*

Proof. Let a learner with target-sized memory be given and k be the corresponding constant. Whenever the learner learns has seen examples $x_0, x_1, \ldots, x_n$ when $|mem_{n+1}| \leq |d| + k$ for all languages L_d which contain the data observed so far. Let x_m be the longest datum seen so far. Let e be the length-lexicographically first index with $\{x_0, x_1, \ldots, x_n\} \subseteq L_e \cup \{\#\}$. If e is shorter than some of the data then

$|mem_{n+1}| \leq |x_m| + k$. Otherwise let e' be the prefix of e of length $|x_m|$.

Consider the dfa which recognises the set $\{conv(d, x) : x \in L_d\}$. Let C be the set of those states which the automata takes on any $u \in L_e$ with $|u| \leq |e'|$ after having processed $conv(e', u)$; it is clear that the automaton will accept $conv(e, u)$ iff it is in a state in C after processing $conv(e', u)$. Hence one can define an automatic function f_C such that $f_C(d')$ is the length-lexicographically least index $d \in I$ such that

$$\forall u \in \Sigma^* \text{ with } |u| \leq |d'|$$
$$[u \in L_d \Leftrightarrow \text{the dfa has after processing } conv(d', u) \text{ a state in the set } C]$$

Now $f_C(d') \leq |d'| + k_C$ for some constant k_C and all d' where $f_C(d')$ is defined. Let k' be the maximum of all $k_{C'}$ where C' ranges over sets of states of the dfa. Furthermore, as $f_C(e')$ is defined and equal to e, one gets that $|e| \leq |f_C(e')| \leq |e'| + k' = |x_m| + k'$ and $|mem_{n+1}| \leq |e| + k \leq |x_m| + k + k'$. The constant $k + k'$ is independent of the language to be learnt and the text selected to present the data; hence the learner has word-sized memory on all texts belonging to languages in the class. ∎

Remark 19.13. The result can be strengthed by saying whenever a class is learnable with target-size memory then it is also learnable with word-size memory. Here the strengthening is that the learner keeps the memory bound also on texts which are for languages outside the class to be learnt.

For this, given an original learner having the word-size memory bound only on languages in the class (with a constant k), one can make a new learner which either has as memory $conv(mem_n, x_m)$ where mem_n is the memory of the original learner and x_m is the longest word seen so far or it has a special value ?. The initial memory is $conv(mem_0, \#)$ and on word x_n it is updated from $conv(mem_n, x_m)$ according to that case which applies:

1. $conv(mem_{n+1}, x_m)$ if $|x_n| < |x_m|$ and $|mem_{n+1}| \leq |x_m| + k$;
2. $conv(mem_{n+1}, x_n)$ if $|x_n| \geq |x_m|$ and $|mem_{n+1}| \leq |x_n| + k$;
3. ? if $|mem_{n+1}| > \max\{|x_m|, |x_n|\} + k$.

Here mem_{n+1} is the memory computed from mem_n and x_n according to the original learner. The hypothesis e_{n+1} of the original learner is taken over in the case that the new memory is not ? and the hypothesis is ? in the case that the new memory is also ?. Note that the special case of the memory and hypothesis being ? only occurs if the original learner violates the word-size memory constraint and that only occurs in the case that the text $x_0, x_1, x_2, \ldots$ is not for a language in the class to be learnt.

Exercise 19.14. *Assume that $\{L_d : d \in I\}$ is the class to be learnt and that every language in the class is finite and that for every language in the class there is exactly* *one index in I. Show that if there is a learner using word-sized memory for this class, then the memory of the same learner is also target-sized. For this, show that there is a constant k such that all $d \in I$ and $x \in L_d$ satisfy $|x| \leq |d| + k$ and then deduce the full result.*

Exercise 19.15. *Show that there is an automatic family $\{L_d : d \in I\}$ such that I contains for each L_d exactly one index and the L_d are exactly the finite subsets of $\{0\}^*$ with even cardinality. Show that the class $\{L_d : d \in I\}$ has an iterative learner using the given hypothesis space. Is the same possible when the class consists of all subsets of $\{0\}^*$ with 0 or 3 or 4 elements? Note that an iterative learner which just conjectured an $d \in I$ must abstain from updating the hypothesis on any datum $x \in L_d$.*

20 Open Problems in Automata Theory

This chapter gives an overview of open problems in automata theory. First some problems left open from research here in Singapore are given, afterwards more difficult, generally open questions are presented.

First: Open Problems from Work in Singapore. There are various open questions related to the memory-usage of automatic learners. These questions have not been solved in the past four years of research on automatic learning. The learners below are always understood to be automatic.

Open Problem 20.1.

1. *Does every automatic family which has an automatic learner also have a learner with word-sized memory?*

2. *Does every automatic family which has a learner with hypothesis-sized memory also have a learner with word-sized memory?*

3. *Does every automatic family which has a learner with hypothesis-sized memory also have an iterative learner?*

In recursion-theory and complexity theory, one often looks at reducibilities which compare sets with functions, for example one has relations like

$$A \leq_m B \Leftrightarrow \exists f \, \forall x \, [A(x) = B(f(x))]$$

where the possible f are taken from a specific class. One could for example do the same with automatic functions. These notions can be refined, for example one can additionally ask that f has to be one-one. Then there are quite trivial examples of incomparable sets: When one fixes the alphabet $\{0,1\}$ then $\{0\}^*$ and $0^*1 \cdot \{0,1\}^*$ are incomparable, as either an exponential set has to be one-one mapped into a linear-sized one or the exponential complement of a set has to be one-one mapped into the linear-sized complement of another set. Both cannot be done, as the image of an exponentially growing set under an automatic function is again exponentially growing. For this, recall that a set A is linear-sized iff there is a linear function f such that A has at most $f(n)$ elements shorter than n; similarly, one can define when A is polynomial-sized and exponential-sized. Wai Yean Tan [73] worked with a slightly modified version where he ignores the alphabet and defines the notions just restricted to the sets to be compared.

Definition 20.2. *Let $A \leq_{au} B$ denote that there is an automatic function f such that*

$$\forall x, y \in A\,[f(x) \neq f(y) \wedge f(x) \in B].$$

Similarly one writes $A \leq_{tr} B$ for the corresponding definition where f is any function computed by a finite transducer.

Wai Yean Tan [73] investigated both notions. For his findings, one needs the following notions: A set A has size $\Theta(n^k)$ iff there a constant c such that up to length n there are at least $n^k/c - c$ and at most $n^k \cdot c + c$ elements in A. A regular set is polynomial-sized in the case that it has size $\Theta(n^k)$ for some k; a regular set is exponential-sized in the case that there is a constant c such that A has at least $2^{n/c} - c$ elements up to length n for each n. Note that every regular set is either finite or polynomial-sized or exponential-sized.

Theorem 20.3. *Let A, B be regular sets.*

1. *The sets A, B are comparable for tr-reducibility: $A \leq_{tr} B$ or $B \leq_{tr} A$. Furthermore, $A \leq_{tr} B$ if one of the following conditions holds:*
   - *A, B are both finite and $|A| \leq |B|$;*
   - *A is finite and B infinite;*
   - *A has size $\Theta(n^k)$ and B has size $\Theta(n^h)$ with $k \leq h$;*
   - *B is exponential-sized.*
2. *If A is polynomial-sized or finite then $A \leq_{au} B$ or $B \leq_{au} A$. If A is of size $\Theta(n^k)$, B is of size $\Theta(n^h)$ and $k < h$ then $A \leq_{au} B$ and $B \not\leq_{au} A$.*

Exercise 20.4. *Make an automatic one-one function which maps the domain $A = 0^*(1^* \cup 2^*)$ to a subset of $B = (0000)^*(1111)^*(2222)^*$, that is, show that $A \leq_{au} B$.*

The question on whether exponential-sized regular sets are always comparable with respect to *au*-reducibility was left open and is still unresolved.

Open Problem 20.5: Tan [73]**.** *Are there regular sets A, B such that $A \not\leq_{au} B$ and $B \not\leq_{au} A$?*

This open problem can be solved in the case that one considers context-free languages in place of regular languages. Then $A = \{x \cdot 2 \cdot y : x, y \in \{0,1\}^*$ and $|x| = |y|\}$ and $B = \{0\}^*$. There is no automatic function mapping A to B in a one-one way, as A is exponential-sized and B is linear-sized. There is no automatic function mapping B to

A in a one-one way, as the range of this function would be an infinite regular set and all words in the language would have exactly one 2 in the middle which contradicts the regular pumping lemma. Hence these sets A and B are incomparable with respect to au-reducibility.

One might also look at reducibilities which are not automatic but still sufficiently easy. One of them is the self-concatenation mapping x to xx. There are two open questions related to this reduction.

Open Problem 20.6: Zhang [78]**.**

1. *Given a regular language A, is there a regular language B such that, for all x, $A(x) = B(xx)$?*
2. *Given a context-free language A, is there a context-free language B such that, for all x, $A(x) = B(xx)$?*

The converse direction is well-known, see, for example, Zhang [78]: If B is regular then the set $A = \{x : xx \in B\}$ is also regular. However, the set $B = \{0^n1^n2^m0^m1^k2^k : n, m, k \in \mathbb{N}\}$ is context-free while the corresponding A given as

$$A = \{x : xx \in B\} = \{0^n1^n2^n : n \in \mathbb{N}\}$$

is not context-free; A is a standard example of a properly context-sensitive set.

Follow-up work by Fung [27] deals with the xm-reducibility. Here one maps x to $x \cdot x^{mi}$ where the function $x \mapsto x^{mi}$ maps an x to its mirror-image, so $(01122123)^{mi} = 32122110$. Now one can show that for every regular set A there is a regular set B such that $A(x) = B(x \cdot x^{mi})$. The set B is chosen as $\{u :$ there are an odd number of pairs (y, z) with $u = yz$ and $y \in A$ and $z \in A^{mi}\}$.

An *ordered group* $(G, +, <, 0)$ satisfies besides the group axioms also the order axioms, namely that $x < y \wedge y < z$ implies $x < z$ and that always exactly one of the three options $x < y$, $y < x$ and $x = y$. Furthermore, the group operation $+$ has to be compatible with the ordering $<$, that is, if $x < y$ then $x+z < y+z$ and $z+x < z+y$ for all x, y, z. Jain, Khoussainov, Stephan, Teng and Zou [38] showed that an automatic ordered group is always commutative. Furthermore, they investigated the following problem which was first posed by Khoussainov.

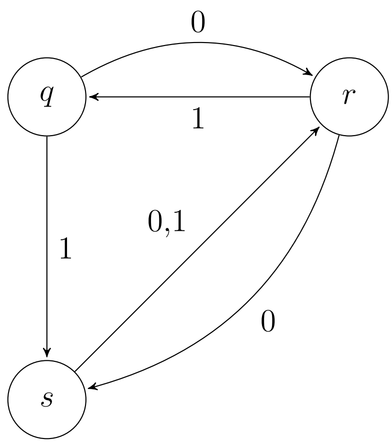
Open Problem 20.7: Khoussainov [38]**.** *Is there an automatic group $(G, +)$ isomorphic to the integers such that $A = \{x \in G : x$ is mapped to a positive number by the isomorphism$\}$ is not regular?*

Jain, Khoussainov, Stephan, Teng and Zou [38] showed that the corresponding question can be answered positively if one takes G to be isomorphic to the rationals with denominators being powers of 6: $G = \{n/6^m : n \in \mathbb{Z} \wedge m \in \mathbb{N}\}$. In this case one can represent the fractional parts as a sum of a binary represented part $n'/2^{m'}$ and ternary represented part $n''/3^{m''}$ and one can do addition on such a representation but one cannot compare the numbers with a finite automaton.

Second: Famous Open Problems. For a given dfa, a synchronising word w such that for all states q, the resulting state $\delta(q, w)$ is the same. Not every dfa has a synchronising word, for example the dfa which computes the remainder by 3 of a sequence of digits cannot have such a state. Černý investigated under which conditions a dfa has a synchronising word and if so, what the length of the shortest synchronising word is. He got the following main result.

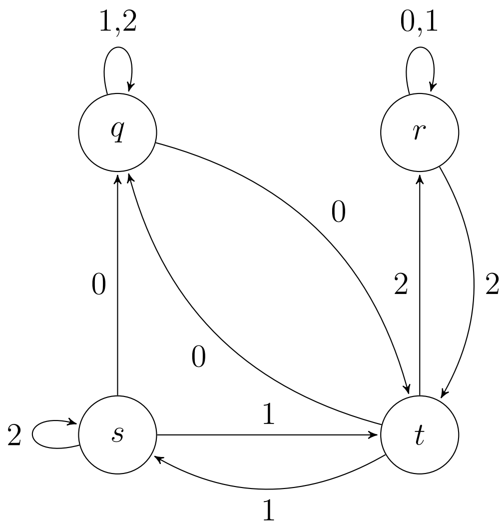
Theorem 20.8: Černý [14]**.** *For each n there is a complete dfa with n states which has a synchronising word of length* $(n-1)^2$ *and no shorter ones.*

Example 20.9. The following automaton gives a dfa for which synchronising words exist and have at least the length 4; note that it is not needed to designate any states as starting or accepting, as this does not matter for the question investigated.



Now the word 0110 is a synchronising word which sends all states to r. For ease of notation, let $\delta(Q, w) = \{\delta(p, w) : p \in Q\}$ for any set Q of states. Note that $\delta(\{q, r, s\}, 1) = \{q, r, s\}$, hence the shortest synchronising word has to start with 0. Now $\delta(\{q, r, s\}, 0) = \{r, s\}$. Note that $\delta(\{r, s\}, 0) = \{r, s\}$, hence the next symbol has to be a 1 in order to achieve something and the synchronising word starts with 01 and $\delta(\{q, r, s\}, 01) = \{q, r\}$. As $\delta(\{q, r\}, 1) = \{q, s\}$ and $\delta(\{q, r\}, 0) = \{r, s\}$, there is no synchronising word of length 3. However, $\delta(\{q, r, s\}, 0110) = \delta(\{q, s\}, 0) = \{r\}$ and 0110 is a shortest synchronising word.

The next example is a complete dfa with $n = 4$ and alphabet $\{0, 1, 2\}$ for which a synchronising word exist and each such word has at least length 9.



The synchronising word for this automaton is 012020120. Again it starts with a 0 and the next symbol has to be a 1 as all others leave the set of reached states the same. The next symbol must be a 2, as a 1 or 0 would undo the modification brought by 01, that is, $\delta(\{q,r,s,t\},010) = \delta(\{q,r,s,t\},011) = \delta(\{q,r,s,t\},0)$. After 012 one can again apply 0 in order to reduce the number of alive states to two: $\delta(\{q,r,s,t\},0120) = \{q,t\}$. Now the next two symbols are 20 in order to move one alive state away from q and one gets $\delta(\{q,t\},20) = \{r,t\}$. Now $\delta(\{r,t\},12) = \{s,t\}$ which is the only set of two alive states which can be mapped into one alive state. This is done by applying 0, so that in summary $\delta(\{q,r,s,t\},012020120) = \{q\}$.

Upper bounds on the length of the shortest synchronising word are also known, however most likely they are not optimal and there is still a considerable gap between the quadratic lower and cubic upper bound.

Theorem 20.10: Frankl [26]; Klyachko, Rystsov and Spivak [47]; Pin [60]. *Assume a complete dfa has n states and has a synchronising word. Then it has a synchronising word not longer than $(n^3 - n)/6$.*

In the following, a weaker form of this theorem is proven with an easier to prove cubic upper bound; this bound is weaker by a factor 3 plus a term of order $O(n)$. Let Q be the set of states. If one has two states q, r and a word w longer than $n(n+1)/2 + 1$ such that $\delta(\{q,r\},w)$ consists of a single state, then there must be a splitting of w into xyz with $y \neq \varepsilon$ such that either $\delta(\{q,r\},x) = \delta(\{q,r\},xy)$ or $\delta(\{q,r\},x)$ consists of a single state, as there are only $n(n+1)/2$ many different pairs of states. In both cases, $\delta(\{q,r\},xz)$ would also consist of a single state, so that w can be replaced by a shorter word. Therefore one can find, inductively, words

$w_1, w_2, \ldots, w_{n-1}$ such that $\delta(Q, w_1 w_2 \ldots w_m)$ has at most $n - m$ states and each w_m has at most length $n(n+1)/2 + 1$. Then the overall length of the synchronising word is at most $n(n^2 - 1)/2 + n - 1 = (n^3 + n - 2)/2$. For some small n it is known that Černý's Conjecture is true.

Example 20.11. If $n = 3$ and the automaton has a synchronising word, then there is a synchronising word of length up to 4.

Proof. Let q, r, s be the states of the complete dfa. One can choose the first symbol of a synchronising word such that at least two states get synchronised. That is, $\delta(\{q, r, s\}, v) \subseteq \{q, r\}$ for a single-letter word v, where q, r, s are some suitable naming of the three states of the dfa. Now there are only three sets of two states, hence each set of two states reachable from $\{q, r\}$ can be reached in up to two symbols. Therefore, a shortest synchronising word w for $\{q, r\}$ must have the property that no set of states is repeated and therefore w has at most the length 3, that is, after the third symbol the corresponding set of alive states has only one element. Thus $\delta(\{q, r, s\}, vw)$ has one element and $|vw| \leq 4$. ∎

One can also show the conjecture for other small values of n; however, the full conjecture is still open.

Open Problem 20.12: Černý's Conjecture [14]. *Černý conjectured that if a complete dfa with n states has synchronising words, then the shortest such word has at most length $(n-1)^2$.*

Exercise 20.13. *Prove Černý's conjecture for $n = 4$; that is, prove that given a complete dfa with four states which has a synchronising word, the shortest synchronising word for this dfa has at most the length 9.*

Another basic question in automata theory is that of the star height. If one permits only the basic operations of forming regular expressions, namely union, concatenation and Kleene star, one can introduce levels of star usage. Namely one does the following:

- Let S_0 contain all finite languages, note that S_0 is closed under union and concatenation;
- For each n, let S_{n+1} contain all languages which can be formed by taking unions and concatenations of languages of the form L or L^* with $L \in S_n$.

The star-height of a regular language L is the minimal n such that $L \in S_n$. Here are some examples.

- The language $L_0 = \{0, 11, 222, 3333\}$ is finite and has star-height 0;
- The language $L_1 = \{00, 11\}^*$ has star-height 1;
- The language $L_2 = (\{00, 11\}^* \cdot \{22, 33\} \cdot \{00, 11\}^* \cdot \{22, 33\})^*$ has star-height 2.

Eggan [21] investigated the star-height and provided a method to compute it from the possible nfas which recognise a regular language. It is known that there are infinitely many different levels of star-height a regular language can take. There is a generalisation, called the generalised star-height. A language is called star-free if it can be build from finite languages and Σ^* using union, intersection, set difference and concatenation. These languages are also called those of generalised star-height 0. The languages of generalised star-height $n+1$ are formed by all expressions obtained by starting with languages of star-height n and their Kleene star languages and then again combining them using union, intersection, set-difference and concatenation. Here examples for the first two levels:

- The language $\{0, 1\}^*$ has generalised star-height 0, as

$$\{0,1\}^* = \Sigma^* - \bigcup_{a \in \Sigma - \{0,1\}} \Sigma^* a \Sigma^*;$$

- L_2 from above has generalised star-height 1, as

$$L_2 = \{00, 11, 22, 33\}^* \cap \{0, 1\}^* \cdot (\{22, 33\} \cdot \{0, 1\}^* \cdot \{22, 33\} \cdot \{0, 1\}^*)^*$$

  and so L_2 is the intersection of two languages of generalised star-height 1;
- $L_3 = \{w : w$ does not have a substring of the form $v\}$ for a fixed v is of generalised star-height 0 as $L_3 = \Sigma^* - \Sigma^* \cdot v \cdot \Sigma^*$;
- $L_4 = \{w : w$ has an even number of $0\}$ is of generalised star-height 1.

It is unknown whether every regular language falls into one of these two levels.

Open Problem 20.14. *Are there any regular languages of generalised star-height 2? Is there a maximal n such that regular languages of generalised star-height n exist? If so, what is this n?*

Exercise 20.15. *Determine the generalised star-height of the following languages over the alphabet* $\{0, 1, 2\}$ *– it is zero or one:*

1. $\{00, 11, 22\}^* \cdot \{000, 111, 222\}^*$;

2. $\{0,1\}^* \cdot 2 \cdot \{0,1\}^* \cdot 2 \cdot \{0,1\}^*$;
3. $(\{0,1\}^* \cdot 2 \cdot \{0,1\}^* \cdot 2 \cdot \{0,1\}^*)^*$;
4. $(\{0,1\}^* \cdot 2 \cdot \{0,1\}^*)^*$;
5. $(\{0,1\}^+ \cdot 22)^*$;
6. $(\{0,1\}^* \cdot 22)^*$;
7. $(((00)^+ \cdot 11)^+ \cdot 22)^+$.

In Thurston automatic groups one selects a subset G of words over the generators to represent all group elements. However, one mostly ignores the words not in G. A central question is how difficult the word problem is, that is, how difficult is it to determine whether a word over the generators (including the inverses) represents a word $w \in G$. That is, if Σ denotes the generators then the word problem is the set $\{(v, w) : v \in \Sigma^*, w \in G, v = w \text{ as group elements}\}$. One can show that the word problem can be solved in polynomial time (PTIME) by the algorithm which starts with the memory u being initialised as the neutral word ε and then reads out one symbol a after another from v and updates u to the member of G representing $u \cdot a$; these updates are all automatic and one has to just invoke the corresponding automatic function $|v|$ times. There is a complexity class LOGSPACE in which one permits the algorithm to use a work space of size logarithmic in the length of the input and to access the input with pointers pointing on some positions and permitting to read the symbol where they point to. These pointers can move forward or backward in the input, but not be moved beyond the beginning and end of the input. Now the algorithm can run arbitrary long but has to keep the memory constraint and at the end comes up with the answer ACCEPT or REJECT. Although there is no time constraint, one can show that the algorithm either needs polynomial time or runs forever, hence LOGSPACE is a subclass of PTIME. An open problem is whether the word problem of a Thurston automatic group can be solved in this subclass.

Open Problem 20.16. *Is the word problem of each Thurston automatic group solvable in LOGSPACE?*

Note that a negative answer to this problem would prove that LOGSPACE $\neq$ PTIME what might even be a more difficult open problem. On the other hand, a positive answer, that is, a LOGSPACE algorithm might be difficult to find, as people looked for it in vane for more than 30 years. So this could be a quite hard open problem.

Widely investigated questions in automata theory is the complexity of membership for the various levels of the Chomsky hierarchy. While for the level of regular language, the usage of dfa provides the optimal answer, the best algorithms are not yet known for the context-free and context-sensitive languages.

Open Problem 20.17. *What is the best time complexity to decide the membership of a context-free language?*

Open Problem 20.18. *Can the membership in a given context-sensitive language be decided in deterministic linear space?*

Both questions are algorithmically important. Cocke, Younger and Kasami provided an algorithm which run in $O(n^3)$ to decide the membership of context-free languages. Better algorithms were obtained using fast matrix multiplication and today bounds around $O(n^{2.38})$ are known. Concerning the context-sensitive membership problem, it is known to be possible in $O(n^2)$ space and non-deterministically in linear space; so the main question is whether this trade-off cen be reduced. These two problems are also quite hard, as any progress which involves the handling of fundamental complexity classes.

One topic much investigated in theoretical computer science is whether the isomorphism problem of certain structures are decidable and this had also been asked for automatic structures. For many possible structures, negative answers were found as the structures were too general. For example, Kuske, Liu and Lohrey [48] showed that it is undecidable whether two automatic equivalence relations are isomorphic. On the other hand, it is decidable whether a linear ordered set is isomorphic to the rationals: By the Theorem of Khoussainov and Nerode, one can decide whether sentences formulated using the ordering in first order logic are true and therefore one checks whether the following conditions are true: (a) There is no least element; (b) There is no greatest element; (c) Between any two elements there is some other element. If these are true, the corresponding linear order is dense and without end-points and therefore isomorphic to the ordering of the rationals, as automatic linear orders have always an at most countable domain. There are still some isomorphism problems for which it is not known whether they can be decided.

Open Problem 20.19. *Are there algorithms which decide the following questions, provided that the assumptions are met?*

1. *Assume that* $(A, Succ_A, P_A)$ *and* $(B, Succ_B, P_B)$ *are automatic structures such that* $(A, Succ_A)$ *and* $(B, Succ_B)$ *are isomorphic to the natural numbers with successor and that* P_A *and* P_B *are regular predicates (subsets) on* A *and* B*. Is* $(A, Succ_A, P_A)$ *isomorphic to* $(B, Succ_B, P_B)$*?*

2. *Assume that* $(A, +)$ *and* $(B, +)$ *are commutative automatic groups. Is* $(A, +)$ *isomorphic to* $(B, +)$*?*

An important open problem for parity games is the time complexity for finding the winner of a parity game, when both players play optimally; initially the algorithms took exponential time [53, 79]. Subsequently Petersson and Vorobyov [59] devised a subexponential randomised algorithm and Jurdziński, Paterson and Zwick [43] a deterministic algorithm of similar complexity; here the subexponential complexity was approximately $n^{O(\sqrt{n})}$. Furthermore, McNaughton [53] showed that the winner of a parity game can be determined in time $O(n^m)$, where n is the number of nodes and m the maximum value aka colour aka priority of the nodes. The following result provides an improved subexponential bound which is also in quasipolynomial time. For the below, it is assumed that in every node, a move can be made, so that the parity game never gets stuck. Furthermore, $\log(h) = \min\{k \in \{1, 2, 3, \ldots\} : 2^k \geq h\}$, so that the logarithm is always a non-zero natural number, what permits to use the logarithm in multiplicative expressions without getting 0 as well as indices in arrays.

Theorem 20.20: Calude, Jain, Khoussainov, Li, Stephan [10]. *One can decide in alternating polylogarithmic space which player has a winning strategy in a given parity game. When the game has n nodes and the values of the nodes are a subset of* $\{1, 2, \ldots, m\}$ *then the algorithm can do this in* $O(\log(n) \cdot \log(m))$ *alternating space.*

Proof. The idea of the proof is that the players move around a marker in the game as before; however, together with the move they update two winning statistics, one for Anke and one for Boris, such that whenever one player follows a memoryless winning strategy for the parity game then this winning statistic will mature (indicating a win for the player) while the winning statistic of the opponent will not mature (and thus not indicate a win for the opponent). It is known that every parity game has for one player a memoryless winning strategy, that is, the strategy tells the player for each node where to move next, independent of the history. The winning statistic of Anke has the following goal: to track whether the game goes through a cycle whose largest node is a node of Anke. Note that if Anke follows a memoryless winning strategy then the game will eventually go through a cycle and the largest node of any cycle the game goes through is always a node of Anke's parity; it will never be a node of Boris' parity, as then Anke's strategy would not be a memoryless winning strategy and Boris could repeat that cycle as often as he wants and thus obtain that a node of his parity is the limit superior of the play.

The naive method to do the tracking would be to archive the last $2n + 1$ nodes visited, however, this takes $O(n \cdot \log(n))$ space and would be too much for the intended result. Thus one constructs a winning statistic which still leads to an Anke win in the case that Anke plays a memoryless winning strategy, however, it will take longer time until it verifies that there was a loop with an Anke-node as largest member, as the winning statistic only memorises partial information due to space restrictions.

Nodes with even value are called Anke-nodes and nodes with an odd value are called Boris-nodes. For convenience, the following convention is made: when comparing nodes with "$<$" and "$\leq$", the corresponding comparison relates to the values of the nodes; when comparing them with "$=$" or "$\neq$", the corresponding comparison refers to the nodes themselves and different nodes with the same value are different with respect for this comparison. Furthermore, the value 0 is reserved for entries in winning statistics which are void and $0 < b$ for all nodes b.

In Anke's winning statistics, an i-sequence is a sequence of nodes $a_1, a_2, \ldots, a_{2^i}$ which had been observed within the course of the game such that, for each $k \in \{1, 2, \ldots, 2^i - 1\}$, the value $\max_{\leq}\{b : b = a_k \vee b = a_{k+1} \vee b$ was observed between a_k and $a_{k+1}\}$ has Anke's parity. For each i-sequence, the winning statistic does not store the sequence itself but it only stores the maximum value b_i of a node which either occurs as the last member of the sequence or occurs after the sequence.

The following invariants are kept throughout the game and are formulated for Anke's winning statistic, those for Boris' winning statistic are defined with the names of Anke and Boris interchanged:

- Only b_i with $0 \leq i \leq \log(n) + 3$ are considered and each such b_i is either zero or a value of an Anke-node or a value of a Boris-node;
- An entry b_i refers to an i-sequence which occurred in the play so far iff $b_i > 0$;
- If b_i, b_j are both non-zero and $i < j$ then $b_i \leq b_j$;
- If b_i, b_j are both non-zero and $i < j$ then they refer to an i-sequence and an j-sequence, respectively, and, in the play of the game, the i-sequence starts only after the value b_j was observed at or after the end of the j-sequence.

Both players' winning statistics are initialised with $b_i = 0$ for all i when the game starts. In each cycle, when the player whose turn is to move has chosen to move into the node with value b, the winning statistics of Anke and then of Boris are updated as follows, here the algorithm for Anke is given and it is followed by an algorithm for Boris with the names of the players interchanged everywhere.

- If b is either an Anke-node or $b > b_0$ then one selects the largest i such that
  (a) either b_i is not an Anke-node but all b_j with $j < i$ are Anke nodes and $(i > 0 \Rightarrow \max\{b_0, b\}$ is an Anke-node)
  (b) or $0 < b_i < b$

  and one updates $b_i = b$ and $b_j = 0$ for all $j < i$;
- If this update produces a non-zero b_i for any i with $2^i > 2n$ then the game terminates with Anke being declared winner.

The winning statistic of Boris is maintained and updated by the same algorithm, with the roles of Anke and Boris being interchanged in the algorithm. When both winning statistics are updated without a termination then the game goes into the next round by letting the corresponding player choose a move.

When updating Anke's winning statistic and the update can be done by case (a) then one can form a new i-sequence by putting the j-sequences for $j = i-1, i-2, \ldots, 1, 0$ together and appending the one-node sequence b which then has the length $2^i = 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0 + 1$; in the case that $i = 0$ this condition just says that one forms a 0-sequence of length 2^0 just consisting of the node b. Note that in the case $i > 0$ the value $\max\{b_0, b\}$ is an Anke-node and therefore the highest node between the last member a of the F_0-sequence and b has the value $\max\{b_h, b\}$ and is an Anke-node. Furthermore, for every $j < i-1$, for the last node a of the $j+1$-sequence and the first node a' of the j-sequence in the new i-sequence, the highest value of a node in the play between these two nodes a, a' is b_{j+1} which, by choice, has Anke's parity. Thus the overall combined sequence is an i-sequence replacing the previous sequences and b is the last node of this sequence and thus, currently, also the largest node after the end of the sequence. All j-sequences with $j < i$ are merged into the new i-sequence and thus their entries are set back to $b_j = 0$.

When updating Anke's winning statistic and the update can be done by case (b) then one only replaces the largest value at or after the end of the i-sequence (which exists by $b_i > 0$) by the new value $b > b_i$ and one discards all j-sequences with $j < i$ what is indicated by setting $b_j = 0$ for all $j < i$.

The same rules apply to the updates of Boris' winning statistics with the roles of Anke and Boris interchanged everywhere.

Note when updating Anke's winning statistic with a move to an Anke-node b, then one can always make an update of type (a) with i being the least number where b_i is not an Anke-node (which exists as the game would have terminated before otherwise). Similarly for updating Boris winning statistics.

If a player wins then the play contains a loop with its maximum node being a node of the player: Without loss of generality assume this winning player to be Anke. The game is won by an i-sequence being observed in Anke's winning statistics with $2^i > 2n$; thus some node occurs at least three times in the i-sequence and there are $h, \ell \in \{1, 2, \ldots, 2^i\}$ with $h < \ell$ such that the same player moves at a_h and a_ℓ and furthermore $a_h = a_\ell$ with respect to the nodes $a_1, a_2, \ldots, a_{F_i}$ of the observed i-sequence. The maximum value b' between a_h and a_ℓ in the play is occurring between some a_k and a_{k+1} (inclusively) for a k with $h \leq k < \ell$. Now, by definition of an i-sequence, b' has Anke's parity. Thus a loop has been observed for which the maximum node is an Anke node.

A player playing a memoryless winning strategy for parity games does not lose: If a player plays a memoryless winning strategy then the opponent cannot go into a loop where the maximum node is of the opponent's parity, as otherwise the opponent could cycle in that loop forever and then win the parity game, contradicting to the player playing a memoryless winning strategy. Thus, when a player follows a memoryless winning strategy, the whole play does not contain any loop where the opponent has the maximum node and so the opponent is during the whole play never declared to be the winner by the winning statistics.

A player playing a memoryless winning strategy for parity games will eventually win: For brevity assume that the player is Anke, the case of Boris is symmetric. The values b_i analysed below refer to Anke's winning statistic.

Assume that an infinite play of the game has the limit superior c which, by assumption, is an Anke-node. For each time t let

$$card(c,t) = \sum_{k:\ b_k(t) \text{ is an Anke-node and } b_k(t) \geq c} 2^k$$

where the $b_k(t)$ refer to the value of b_k at the end of step t. Now it is shown that whenever at times t, t' with $t < t'$ a move to c was made with c being an Anke-node and no move strictly between t, t' was to any node $c' \geq c$ then $card(c,t) < card(c,t')$. To see this, let i be the largest index where there is a step t'' with $t < t'' \leq t'$ such that b_i becomes updated in step t''. Now one considers several cases:

- Case $b_i(t'') = 0$: This case does only occur if also b_{i+1} gets updated and contradicts the choice of i, so it does not need to be considered.
- Case $b_i(t) \geq c$ and $b_i(t)$ is an Anke node: In this case, the only way to update this node at t'' is to do an update of type (a) and then also the entry $b_{i+1}(t'')$ would be changed in contradiction of the choice of i, so this case also does not need to be considered.
- Case $b_i(t)$ is a Boris node and $b_i(t) \geq c$: Then an update is possible only by case (a). If $b_i(t'') < c$ then, at step t', another update will occur and enforce by (b) that $b_i(t') = c$. The value $card(c,t)$ is largest when all $b_j(t)$ with $j < i$ are Anke-nodes at step t and even in this worst case it holds that $card(c,t') - card(c,t) \geq 2^i - \sum_{j:j<i} 2^j \geq 1$.
- Case $0 < b_i(t) < c$: Then latest at stage t', as an update of type (b) at i is possible, it will be enforced that $b_i(t') = c$ while $b_j(t) < c$ for all $j \leq i$ and therefore $card(c,t') \geq card(c,t) + 2^i \geq card(c,t) + 1$.
- Case $b_i(t) = 0$: Then at stage t'' an update of type (a) will make $b_i(t'') > 0$ and, in the case that $b_i(t'') < c$, a further update of type (b) will at stage t' enforce that $b_i(t') = c$. Again, the value $card(c,t)$ is largest when all $b_j(t)$ with $j < i$ are Anke-nodes at step t and even in this worst case it holds that $card(c,t') - card(c,t) \geq 2^i - \sum_{j:j<i} 2^j \geq 1$.

Thus, once all moves involving nodes larger than c have been done in the play, there will still be infinitely many moves to nodes of value c and for each two subsequent such moves at t, t' it will hold that $card(c, t) + 1 \leq card(c, t')$. As a consequence, the number $card(c, t)$ for these nodes will, for sufficiently large t where a move to c is made, rely on some i with $b_i(t) \geq c$ and $2^i > 2n$ and latest then the termination condition of Anke will terminate the game with a win for Anke.

Thus, an alternating Turing machine can simulate both players and it will accept the computation whenever Anke has a winning strategy for the game taking the winning statistics into account. Thus the alternating Turing machine with space usage of $O(\log(n) \cdot \log(m))$ can decide whether the game, from some given starting point, will end up in Anke winning or in Boris winning, provided that the winner plays a memoryless winning strategy for the corresponding parity game (which always exists when the player can win the parity game). ∎

Chandra, Kozen and Stockmeyer [13] showed that everything what can be computed by an alternating Turing machine in polylogarithmic space can also be computed deterministically in quasipolynomial time. More precisely, their more precise bounds give that the running time of a deterministic Turing machine for the above mentioned problem is $O(n^{c\log(m)})$ for some constant c.

Theorem 20.21: Calude, Jain, Khoussainov, Li, Stephan [10]**.** *Assume that a parity game has n nodes which take values from $\{1, 2, \ldots, m\}$, note that one can always choose $m \leq n + 1$. Now one can decide in time $O(n^{c\log(m)})$ which player has a winning strategy in the parity game.*

In some special cases with respect to the choice of m in dependence of n, one can obtain a polynomial time bound. McNaughton [53] showed that for every constant m, one can solve a parity game with n nodes having values from $\{1, 2, \ldots, m\}$ in time $O(n^m)$; Schewe [68, 70] and others brought down the bound, but it remained dependent on m. The next result shows that for fixed m and large n one can determine the winner of the parity game in $O(n^{5.04})$; the bound is, however, more general: If $m \leq h \cdot \log(n)$ then one can determine the winner of a parity game in $O(h^4 \cdot n^{3.45+\log(h+2)})$. This implies that one can solve the parity games in $O((16n)^{3.45+\log(\lceil m/\log(n)\rceil+2)})$. Calude, Jain, Khoussainov, Li and Stephan [10] give a slightly better bound for $h = 1$.

Theorem 20.22. *If $m \leq h \cdot \log(n)$ and $h \in \mathbb{N}$ then one can solve the parity game with n nodes which have values from $\{1, 2, \ldots, m\}$ in time $O(h^4 \cdot n^{3.45+\log(h+2)})$.*

Proof. Note that Theorem 20.20 actually showed that the following conditions are equivalent:

- Anke can win the parity game;
- Anke can play the parity game such that her winning statistic matures while Boris' winning statistic does not mature.

Thus one can simplify this and play a survival game with the following property: Anke wins the game iff the parity game runs forever without Boris achieving a win according to his winning statistics. If Boris follows a memoryless winning strategy for the parity game then Anke loses, if Anke follows a memoryless winning strategy for the parity game then she wins. Thus it is sufficient to track only Boris' winning statistics for the game. Thus Anke has a winning strategy for the parity game iff she has a winning strategy for the following survival game:

- The set Q of nodes of the survival game consists of nodes of the form $(a, p, \tilde{b})$ where a is a node of the parity game, the player $p \in \{\text{Anke}, \text{Boris}\}$ is that player whose turn is to move next and $\tilde{b}$ represents the winning statistic of Boris;
- Anke can move from $(a, \text{Anke}, \tilde{b})$ to $(a', \text{Boris}, \tilde{b}')$ iff she can move from a to a' in the parity game and this move causes the winning statistic of Boris to be updated from $\tilde{b}$ to $\tilde{b}'$;
- Boris can move from $(a, \text{Boris}, \tilde{b})$ to $(a', \text{Anke}, \tilde{b}')$ iff he can move from a to a' in the parity game and this move causes the winning statistic of Boris to be updated from $\tilde{b}$ to $\tilde{b}'$;
- The starting node is $(s, \text{Anke}, \tilde{0})$ where $\tilde{0}$ is the vector of all b_i being 0 and s is the starting node of the parity game.

To estimate the number of members of Q, first one codes Boris' winning condition $b_0, b_1, \ldots, b_{\lceil\log(n)\rceil+2}$ by a new sequence $\hat{b}_0, \hat{b}_1, \ldots, \hat{b}_{\lceil\log(n)\rceil+2}$ as follows: $\hat{b}_0 = b_0$ and, for all $i < \lceil\log(n)\rceil+2$, if $b_{i+1} = 0$ then $\hat{b}_{i+1} = \hat{b}_i+1$ else $\hat{b}_{i+1} = \hat{b}_i+2+\min\{b_{i+1}-b_j : j \leq i\}$. Note that the latter just says that $b_{i+2} = \hat{b}_i + 2 + (b_i - b_j)$ for the most recent j where $b_j \neq 0$. Now $\hat{b}_{\lceil\log(n)\rceil+2} \leq 2 \cdot (\lceil\log(n)\rceil + 2) + h \cdot b_{\lceil\log(n)\rceil+2} \leq (h+2) \cdot (\lceil\log(n)\rceil + 3)$ what gives $O(n^{h+2})$. Thus the number of possible values of the winning statistics can all be coded with $(h+2) \cdot (\lceil\log(n)\rceil + 3)$ bits. However, one can get a better value by observing that only $\lceil\log(n)\rceil + 3$ of these bits are 1. The number of all ways to choose $\lceil\log(n)\rceil + 3$ out of $(h+2) \cdot (\lceil\log(n)\rceil + 3)$ numbers can, by the Wikipedia page on binomial coefficients and the inequality using the entropy in there, be bounded by

$$\begin{aligned}
& 2^{(\log(n)+4)\cdot(h+2)\cdot((1/(h+2))\cdot\log(h+2)+((h+1)/(h+2))\cdot\log((h+2)/(h+1)))} \\
= \;& 2^{(\log(n)+4)\cdot(\log(h+2)+\log(1+1/(h+2))\cdot(h+1))} \\
= \;& (16n)^{\log(h+2)+(\log(1+1/(h+2))\cdot(h+1))} \\
\leq \;& (16n)^{1.45+\log(h+2)} \leq c \cdot h^4 \cdot n^{1.45+\log(h+2)}
\end{aligned}$$

for some constant c, so the whole expression is in $O(h^4 \cdot n^{1.45+\log(h+2)})$. In these equations, it is used that $\log(1+1/(h+2)) \cdot (h+1) \leq 1.45$ for all $h \in \mathbb{N}$. Furthermore, one has to multiply this by one n and by 2 in order to store the current player and current position, so in total Q has size $O(h^4 \cdot n^{2.45+\log(h+2)})$ and the remaining part of the proof will show that the runtime is bounded by $O(h^4 \cdot n^{3.45+\log(h+2)})$.

The survival game can be decided in $O(|Q| \cdot n)$: The algorithm would be the following: First one computes for each node $q \in Q$ the list of the up to n successors and also generates a linked list of predecessors such that the collection of all these lists together has the length $|Q| \cdot n$. These inverted lists can also be generated in time $O(|Q| \cdot n)$. Furthermore, one can determine a list of $Q' \subseteq Q$ of nodes where Boris winning statistic has matured (that is, Boris has won); determining these nodes is also in time $O(|Q|)$.

Note that a node is a winning node for Boris if either Anke moves from this node and all successor nodes are winning nodes for Boris or Boris moves from this node and some successor is a winning node for Boris. This idea will lead to the algorithm below.

For this, a tracking number k_q is introduced which is maintained such that the winning nodes for Boris will eventually all have $k_q = 0$ and that k_q indicates how many further times one has to approach the node until it can be declared a winning node for Boris. The numbers k_q are initialised by the following rule:

- On nodes $q \in Q'$ the number k_q is 1;
- On nodes $q = (a, \text{Anke}, \tilde{b}) \notin Q'$, the number k_q is initialised as the number of nodes q' such that Anke can move from q to q';
- On nodes $q = (a, \text{Boris}, \tilde{b}) \notin Q'$, the number k_q is initialised as 1;

These numbers can be computed from the length of the list of predecessors of q for each $q \in Q$. Now one calls the following recursive procedure initially for all $q \in Q'$ and each call updates the number k_q. The recursive call does the following:

- If $k_q = 0$ then return without any further action else update $k_q = k_q - 1$;
- If after this update still $k_q > 0$ then return without further action;
- Otherwise, that is when k_q originally was 1 when entering the call then call recursively all predecessors q' of q with the same algorithm.

After the termination of all these recursive calls, one looks at k_q for the start node q of the survival game. If $k_q > 0$ then Anke wins else Boris wins.

Note that in this algorithm, for each node $q \in Q$ the predecessors are only called at most once, namely when k_q goes down from 1 to 0 and that is the time where it is determined that the node is a winning node for Boris. Thus there are at most

$O(|Q| \cdot n)$ many recursive calls and the overall complexity is $O(|Q| \cdot n)$.

For the verification, the main invariant is that k_q originally says for how many of the successors of q one must check that they are winning nodes for Boris until one can conclude that the node q is also a winning node of Boris. In the case that the winning statistics of Boris have matured in the node q, the value k_q is taken to be 1 so that the node is processed once with all the recursive calls in the recursive algorithm. For nodes where it is Boris' turn to move, there needs also be only one outgoing move which produces a win of Boris. Thus one initialises k_q as 1 and as soon as this outgoing node is found, k_q goes to 0 what means that the node is declared a winning node for Boris. In the case that the node q is a node where Anke moves then one has to enforce that Anke has no choice but to go to a winning node for Boris. Thus k_q is initialised as the number of moves which Anke can move in this node and each time when one of these successor nodes is declared a winning node for Boris, k_q goes down by one. Note that once the recursive algorithm is completed for all nodes, exactly the nodes with $k_q = 0$ are the winning nodes of Boris in this survival game. ∎

For the special case of $h = 1$, the more direct bound $O(n^{h+4})$ is slightly better than the derived bound of $O(n^{3.45+\log(3)})$; however, in the general case of larger h, the bound $O(n^{3.45+\log(h+2)})$ is better.

When considering $h = 1$, this special case shows that, for each constant m, the parity game with n nodes having values from $\{1, 2, \ldots, m\}$ can be solved in time $O(n^5) + g(m)$ for some function g. Such problems are called "Fixed Parameter Tractable", as for each fixed parameter m the corresponding algorithm runs in polynomial time and this polynomial is the same for all m, except for the additive constant $g(m)$ depending on m. Downey and Fellows [20] provide an introduction to the field of parameterised complexity.

Exercise 20.23. *Show that one can decide, for all sufficiently large m, n, the parity games with n nodes and values from $\{1, 2, \ldots, m\}$ in time $O(n^{\log(m)+20})$; for this use a direct coding of the winning conditions with $\lceil \log(n) + 3 \rceil \cdot \lceil \log(m) + 1 \rceil$ bits rather than the above methods with the binomial coefficients. Furthermore, show that the memoryless winning-strategy of the winner can then be computed with the same time bound (the constant 20 is generous enough).*

Open Problem 20.24. Is there a polynomial time algorithm (in the number n of nodes of the parity game) to decide which player would win the parity game?

Selftest 20.25. *Construct a learning algorithm for the class of all regular languages which uses only equivalence queries such that the number of queries is linear in the sum consisting of the number of states of a dfa for the target and the number of symbols in the longest counter example seen.*

Selftest 20.26. *Let* $I = \{0,1\}^*$ *and for all* $e \in I$, $L_e = \{x \in \{0,1\}^* : x <_{lex} e\}$. *Is this automatic family learnable from positive data?*

If the answer above is "yes" then describe how the learner works; if the answer above is "no" then explain why a learner does not exist.

Selftest 20.27. *Assume that an automatic representation of the ordinals strictly below* ω^n *is given for some positive natural number* n. *Now consider the class of all sets* $L_{\alpha,\beta} = \{\gamma < \omega^n : \alpha \leq \gamma < \beta\}$, *where* α *and* β *are ordinals chosen such that the set* $L_{\alpha,\beta}$ *is closed under ordinal addition, that is, when* $\gamma, \gamma' \in L_{\alpha,\beta}$ *so is* $\gamma + \gamma'$. *If this class is learnable then provide an automatic an automatic learner (using some automatic family representing the class as hypothesis space) else explain why the class is not learnable.*

Selftest 20.28. *Assume that a dfa has states and alphabet* $\{0, 1, \ldots, 9\}$ *and the successor of state* a *on symbol* b *is defined as follows: If* $a < b$ *then the successor is* $b - a - 1$ *else the successor is* $a - b$.

Determine whether this dfa has a synchronising word, that is, a word which maps all states to the same state. If so then write-down a synchronising word which is as short as possible else explain why there is no synchronising word.

Solution for Selftest 20.25. The idea is to "cheat" by forcing the teacher to output long counterexamples. So one takes a list of all deterministic finite automata dfa_0, dfa_1, ... and computes for each dfa_n and automaton dfa'_n such that

- dfa'_n has at least n states and there is no dfa with less states for the same language;
- the language recognised by dfa'_n differs from the language recognised by dfa_n by exactly one word of at least length n.

This can be achieved by searching for the first $m \geq n$ such that the minimal automaton for the language obtained by taking the symmetric difference of $\{0^m\}$ and the language recognised by dfa_n needs at least n states. The automata dfa'_n can be computed from dfa_n in polynomial time. Now the algorithm does the following:

1. Let $n = 0$;
2. Compute dfa_n and dfa'_n;
3. Ask if dfa'_n is correct;
4. If answer is "yes" then conjecture dfa'_n and terminate;
5. Ask if dfa_n is correct;
6. If answer is "yes" then conjecture dfa_n and terminate;
7. Let $n = n + 1$ and go to step 2.

Note that in this algorithm, if the language to be learnt turns out to be the one generated by dfa'_n then the number of states of the dfa is at least n and only $2n - 1$ queries had been made until learning success; if the language to be learnt turns out to be the one generated by dfa_n then dfa'_n had been asked before, differing from the language of dfa_n by exactly one word which has length n or more and at only $2n$ queries had been made, again the complexity bound is kept.

Solution for Selftest 20.26. The answer is "no". Assume by way of contradiction, that there is a learner M. Then $L_1 = \{\varepsilon\} \cup 0 \cdot \{0,1\}^*$. By Angluin's tell-tale condition, there is a finite subset F of L_1 such that there should be no set L_u with $F \subseteq L_u \subset L_1$. Given such an F, let n be the length of the longest word in F and consider L_{01^n}. All members of L_1 up to length n satisfy that they are lexicographically strictly before 01^n and thus $F \subseteq L_{01^n} \subset L_1$. Thus, F cannot be a tell-tale set for L_1 and the class cannot be learnable by Angluin's tell-tale criterion.

Solution for Selftest 20.27. If $\gamma \in L_{\alpha,\beta}$ then also $\gamma + \gamma$, $\gamma + \gamma + \gamma$ and so on are in $L_{\alpha,\beta}$, furthermore, if $\omega^k \leq \gamma < \omega^{k+1}$ then all numbers between γ and ω^{k+1} (excluding ω^{k+1} itself) must be in $L_{\alpha,\beta}$; in the case that $\gamma = 0$, $\omega^{m(\gamma)} = 1$. Thus β is of the form

ω^m for some m with $0 \leq m \leq n$. So the learning algorithm does the following:

For each datum γ the learner computes $m(\gamma)$ to be the first ω-power strictly above γ, that is, $\gamma < \omega^{m(\gamma)} \leq \omega^m$ for all m with $\gamma < \omega^m$. The learner conjectures $\emptyset$ until some datum $\gamma \neq \#$ is observed. Then the learner let $\alpha = \gamma$ and $\beta = \omega^{m(\gamma)}$. At every further datum γ, α is replaced by $\min\{\alpha, \gamma\}$ and β is replaced by $\max\{\beta, \omega^{m(\gamma)}\}$.

This learner is automatic, as one can choose an automatic structure representing all ordinals up to ω^n together with their order; there are only finitely many ordinals of the form ω^m with $0 \leq m \leq n$, these can be archived and one can just define $\omega^{m(\gamma)}$ to be the least strict upper bound of γ from this finite list. Furthermore, the learner converges to a final hypothesis $L_{\alpha,\beta}$, as the minimum α of the language is seen after finite time and as the strict upper bound β, by being from a finite list, can only be updated a finite number of times to a larger number.

Solution for Selftest 20.28. The dfa has a synchronising word of length 4.

For getting a lower bound on the length, one can see that at most two states are mapped to the same state by a symbol b. So the first symbol maps the given 10 states to at least 5 different states, the next symbol maps these 5 states to at least 3 different states, the third symbol maps these 3 states to at least 2 states and the fourth symbol then might, perhaps, map the two states to one state. So the length of each synchronising word is at least 4.

Now consider the word 5321. The symbol 5 maps the states $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ to $\{0, 1, 2, 3, 4\}$, the symbol 3 maps $\{0, 1, 2, 3, 4\}$ to $\{0, 1, 2\}$, the symbol 2 maps $\{0, 1, 2\}$ to $\{0, 1\}$ and the symbol 1 maps $\{0, 1\}$ to $\{0\}$. Hence 5321 is a shortest synchronising word.

References

[1] Wilhelm Ackermann (1928). Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[2] Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45:117–135, 1980.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[4] Lenore Blum and Manuel Blum. Towards a mathematical theory of inductive inference. *Information and Control*, 28:125–155, 1975.

[5] Henrik Björklund, Sven Sandberg and Sergei Vorobyov. *On fixed-parameter complexity of infinite games.* Technical report 2003-038, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden.

[6] Henrik Björklund, Sven Sandberg and Sergei Vorobyov. Memoryless determinacy of parity and mean payoff games: a simple proof. *Theoretical Computer Science*, 310(1–3):365–378, 2004.

[7] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the Association of Computing Machinery*, 11:481–494, 1964.

[8] J. Richard Büchi. On a decision method in restricted second order arithmetic. *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, Stanford University Press, Stanford, California, 1960.

[9] J. Richard Büchi and Lawrence H. Landweber: Definability in the monadic second order theory of successor. *The Journal of Symbolic Logic*, 34:166–170, 1966.

[10] Cristian Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li and Frank Stephan. *Deciding parity games in quasipolynomial time.* Manuscript, 2016.

[11] John Case, Sanjay Jain, Trong Dao Le, Yuh Shin Ong, Pavel Semukhin and Frank Stephan. Automatic learning of subclasses of pattern languages. *Information and Computation*, 218:17–35, 2012.

[12] Christopher Chak, Rūsiņš Freivalds, Frank Stephan and Henrietta Tan. On block pumpable languages. *Theoretical Computer Science*, 609:272–285, 2016.

[13] Ashok K. Chandra, Dexter C. Kozen and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[14] Jan Černý. Poznámka k homogénnym experimentom s konečnými automatami. *Matematicko-fyzikálny Časopis Slovenskej Akadémie Vied*, 14:208–216, 1964. In Slovak. See also `http://en.wikipedia.org/wiki/Synchronizing_word`.

[15] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[16] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[17] John Cocke and Jacob T. Schwartz. *Programming languages and their compilers: Preliminary notes.* Technical Report, Courant Institute of Mathematical Sciences, New York University, 1970.

[18] Elias Dahlhaus and Manfred K. Warmuth. Membership for Growing Context-Sensitive Grammars Is Polynomial. *Journal of Computer and System Sciences*, 33:456–472, 1986.

[19] Christian Delhommé. Automaticité des ordinaux et des graphes homogènes. *Comptes Rendus Mathematique*, 339(1):5–10, 2004.

[20] Rodney G. Downey and Michael R. Fellows. *Parameterised Complexity.* Springer, Heidelberg, 1999.

[21] Lawrence C. Eggan. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal*, 10(4):385–397, 1963.

[22] Andrzej Ehrenfeucht, Rohit Parikh and Grzegorz Rozenberg. Pumping lemmas for regular sets. SIAM Journal on Computing, 10:536–541, 1981.

[23] Andrzej Ehrenfeucht and Grzegorz Rozenberg. On the separating power of EOL systems. *RAIRO Informatique théorique* 17(1): 13–22, 1983.

[24] Andrzej Ehrenfeucht and H. Paul Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[25] David Epstein, James Cannon, Derek Holt, Silvio Levy, Michael Paterson and William Thurston. *Word Processing in Groups.* Jones and Bartlett Publishers, Boston, Massachusetts, 1992.

[26] Péter Frankl. An extremal problem for two families of sets. *European Journal of Combinatorics* 3:125–127, 1982.

[27] Dennis Fung. Automata Theory: The XM Problem. BComp Dissertation (Final Year Project), School of Computing, National University of Singapore, 2014.

bibitemGLMMO15 Jakub Gajarský, Michael Lampis, Kazuhisa Makino, Valia Mitsou and Sebastian Ordyniak. Parameterised algorithms for parity games. *Mathematical Foundations of Computer Science*, MFCS 2015. *Springer LNCS* 9235:336–347, 2015.

[28] William I. Gasarch. Guest Column: The second P =? NP Poll. *SIGACT News Complexity Theory Column*, 74, 2012.
`http://www.cs.umd.edu/~gasarch/papers/poll2012.pdf`

[29] Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. *ACM Transactions in Computational Logic*, 13(1):4, 2012.

[30] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatshefte fr Mathematik Physik*, 38: 173–198, 1931.

[31] Mark Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[32] Sheila Greibach. A new normal-form theorem for context-free phrase structure grammars. *Journal of the Association of Computing Machinery*, 12(1):42–52, 1965.

[33] Bernard R. Hodgson. *Théories décidables par automate fini.* Ph.D. thesis, University of Montréal, 1976.

[34] Bernard R. Hodgson. Décidabilité par automate fini. *Annales des sciences mathématiques du Québec*, 7(1):39–57, 1983.

[35] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Third Edition, Addison-Wesley Publishing, Reading Massachusetts, 2007.

[36] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[37] Jeffrey Jaffe. A necessary and sufficient pumping lemma for regular languages. *ACM SIGACT News*, 10(2):48–49, 1978.

[38] Sanjay Jain, Bakhadyr Khoussainov, Frank Stephan, Dan Teng and Siyuan Zou. On semiautomatic structures. Computer Science – Theory and Applications – Ninth International *Computer Science Symposium in Russia*, CSR 2014, Moscow, Russia, June 7–11, 2014. Proceedings. Springer LNCS 8476:204–217, 2014.

[39] Sanjay Jain, Qinglong Luo and Frank Stephan. Learnability of automatic classes. *Language and Automata Theory and Applications*, Fourth International Conference, LATA 2010, Trier, May 2010, Proceedings. Springer LNCS 6031:293–307, 2010.

[40] Sanjay Jain, Yuh Shin Ong, Shi Pu and Frank Stephan. On automatic families. *Proceedings of the 11th Asian Logic Conference*, ALC 2009, in Honour of Professor Chong Chitat's 60th birthday, pages 94–113. World Scientific, 2011.

[41] Sanjay Jain, Yuh Shin Ong and Frank Stephan. Regular patterns, regular languages and context-free languages. *Information Processing Letters* 110:1114–1119, 2010.

[42] Marcin Jurdziński. Deciding the winner in parity games is in $\mathbf{UP} \cap \mathbf{Co} - \mathbf{UP}$. *Information Processing Letters*, 68(3):119–124, 1998.

[43] Marcin Jurdziński, Mike Paterson and Uri Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

[44] Tadao Kasami. *An efficient recognition and syntax-analysis algorithm for context-free languages.* Technical Report, Air Force Cambridge Research Laboratories, 1965.

[45] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. *Logical and Computational Complexity*, (International Workshop LCC 1994). Springer LNCS 960:367–392, 1995.

[46] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and its Applications.* Birkhäuser, 2001.

[47] A.A. Klyachko, Igor K. Rostsov and M.A. Spivak. An extremal combinatorial problem associated with the bound on the length of a synchronizing word in an automaton. *Cybernetics and Systems Analysis / Kibernetika*, 23(2):165–171, 1987.

[48] Dietrich Kuske, Jiamou Liu and Markus Lohrey. The isomorphism problem on classes of automatic structures with transitive relations. *Transactions of the American Mathematical Society*, 365:5103–5151, 2013.

[49] Roger Lyndon and Marcel-Paul Schützenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Mathematical Journal*, 9:289–298, 1962.

[50] Yuri V. Matiyasevich. Diofantovost' perechislimykh mnozhestv. *Doklady Akademii Nauk SSSR*, 191:297-282, 1970 (Russian). English translation: Enumerable sets are Diophantine, *Soviet Mathematics Doklady*, 11:354-358, 1970.

[51] Yuri Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, Massachusetts, 1993.

[52] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[53] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[54] George H. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, 34(5):1045–1079, 1955.

[55] Edward F. Moore. Gedanken Experiments on sequential machines. *Automata Studies*, edited by C.E. Shannon and John McCarthy, Princeton University Press, Princeton, New Jersey, 1956.

[56] Anil Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.

[57] Maurice Nivat. Transductions des langages de Chomsky. *Annales de l'institut Fourier*, Grenoble, 18:339–455, 1968.

[58] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2:191–194, 1968.

[59] Viktor Petersson and Sergei G. Vorobyov. A randomized subexponential algorithm for parity games. *Nordic Journal of Computing*, 8:324–345, 2001.

[60] Jean-Éric Pin. On two combinatorial problems arising from automata theory. *Annals of Discrete Mathematics*, 17:535–548, 1983.

[61] Michael O. Rabin and Dana Scott. Finite Automata and their Decision Problems, *IBM Journal of Research and Development*, 3:115–125, 1959.

[62] Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

[63] Henry Gordon Rice. Classes of enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74:358–366, 1953.

[64] Rockford Ross and Karl Winklmann. Repetitive strings are not context-free. *RAIRO Informatique théorique* 16(3):191–199, 1982.

[65] Shmuel Safra. On the complexity of ω-automata. Proceedings twenty-ninth IEEE Symposium on Foundations of Computer Science, pages 319-327, 1988.

[66] Shmuel Safra. Exponential determinization for omega-Automata with a strong fairness acceptance condition. *SIAM Journal on Computing*, 36(3):803–814, 2006.

[67] Walter Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[68] Sven Schewe. Solving parity games in big steps. *FCTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS 4855:449–460, 2007.

[69] Sven Schewe. Büchi Complementation Made Tight. *Symposium on Theoretical Aspects of Computer Science* (STACS 2009), pages 661-672, 2009.

[70] Sven Schewe. From parity and payoff games to linear programming. *Mathematical Foundations of Computer Science 2009*, Thirtyfourth International Symposium, MFCS 2009, Novy Smokovec, High Tatras, Slovakia, August 24-28, 2009. Proceedings. *Springer LNCS*, 5734:675–686, 2009.

[71] Thoralf Skolem. Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Anwendungsbereich. *Videnskapsselskapets Skrifter I, Mathematisch-Naturwissenschaftliche Klasse* 6, 1923.

[72] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the European Association for Theoretical Computer Science*, 96–100, 1987.

[73] Wai Yean Tan. *Reducibilities between regular languages.* Master Thesis, Department of Mathematics, National University of Singapore, 2010.

[74] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936.

[75] Axel Thue. Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. *Norske Videnskabers Selskabs Skrifter, I, Mathematisch-Naturwissenschaftliche Klasse* (Kristiania), 10, 34 pages, 1914.

[76] Alan M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936 and correction, 43:544–546, 1937.

[77] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:198–208, 1967.

[78] Jiangwei Zhang. *Regular and context-free languages and their closure properties with respect to specific many-one reductions.* Honours Year Project Thesis, Department of Mathematics, National University of Singapore, 2013.

[79] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200:135–183, 1998.