CS3231 – Theory of Computation

Frank Stephan

Semester I, Academic Year 2021-2022

Theory of Computation is a lecture which will introduce formal languages on all levels of the Chomsky hierarchy and besides the grammar approach also provide with the automata / machine approach to these languages. It will provide a theory of regular and context-free languages as well as basic recursion theory.

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 65162759 and 65164246
Email fstephan@comp.nus.edu.sg
Homepage http://www.comp.nus.edu.sg/~fstephan/index.html

These lecture notes contain all material relevant for the examinations and the course. For further reading, the author refers to the textbooks *Automata Theory and its Applications* by Bakhadyr Khoussainov and Anil Nerode [54] and *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani and Ullman [43].

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

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 [58]. *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 $j = 4$:

$$u_1\,u_2\,u_3\,u_1\,u_2\,u_3\,u_1\,u_2\,u_3\,u_1\,u_2\,u_3,$$
$$u_1\,u_2\,u_3\,u_4\,u_1\,u_2\,u_3\,u_4\,u_1\,u_2\,u_3\,u_4.$$

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.

Exercise 1.21. *Let L satisfy the following statement: For all $u, v, w \in L$, either $uv = vu$ or $uw = wu$ or $vw = wv$. Which of the following statements are true for all such L:*

(a) *All $x, y \in L$ satisfy $xy = yx$;*
(b) *All sufficiently long $x, y \in L$ satisfy $xy = yx$;*
(c) *The language L has polynomial growth.*

Give an answer to these questions and prove them.

Exercise 1.22. *Let L consist of all words which contain each of the letters* 0, 1, 2, 3 *exactly once. Make a regular expression generating L which has at most length* 100 *for L. For the length of the expression, each digit, each comma, each concatenation symbol and each bracket and each set bracket counts exactly as one symbol. Concatenation binds more than union. Kleene star and plus should not be used, as L is finite.*

Exercise 1.23. *Make a regular expression for the set* $\{w \in \{0\}^* : |w| \leq 9\}$ *which has at most 26 characters using explicit lists of finite sets and concatenation and union. In the expression, the concatenation symbol, the union symbol, each set bracket, each comma, each symbol* 0 *and each symbol* ε*, all count all as one character.*

In the following, let V be the set of vowels, W be the set of consonants, S be the set of punctuation marks and T be the set of spacings (blancs and new lines and so on). Note that all three exercises have slightly different definitions of words, though they always are strings of some vowels and perhaps some consonants. There is no need to distinguish upper and lower case letters.

Exercise 1.24. *Make a regular expression (using above sets) of all words which contain at least two vowels and before, after and between vowels is exactly one consonant. Examples of such words are "woman", "regular", "lower" but not "upper", "man", "runner", "mouse" and "mice".*

Exercise 1.25. *Make a regular expression of all sentences where each sentence consists of words containing one vowel and arbitrarily many consonants and between two words are spacings and after the last word is a punctuation mark. Example: "can dogs and cats run fast?"*

Exercise 1.26. *Make a regular expressions generating texts of sentences separted by spacings where sentences are as above with the only difference that words can have one or two vowels and up to four consonants. Example: "can dogs and cats run very fast? yes, they can. sheep can run, too."*

In the next four exercises, L and H range over infinite subsets of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\} \cdot \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$ which are viewed as sets of natural numbers having their usual value in the decimal system. Furthermore, $p, q \geq 2$ and $p, q \in \mathbb{N}$.

Exercise 1.27. *If* L *does not contain numbers of the form* $x, x + 1$*, the same is true for* $L \cdot L$*. Prove the answer, in the case of a negative answer, provide a regular expression for the regular set* L*.*

Exercise 1.28. *There is a number* p *and some* L *as specified above such that both* L *and* L^+ *consist only of powers of* p*. Either provide* p *and regular expression for* L *or prove that there are no such* p *and* L*.*

Exercise 1.29. *Let* L^q *denote the concatenation of* q *copies of* L*. Find* p, q *such that the following property* $(*)$ *is true:* $(*)$*: Every infinite set* L *has an infinite subset* H *such that* H^q *consists only of numbers divisible by* p*. Prove the answer.*

Exercise 1.30. *Prove that* $(*)$ *from Exercise 1.29 is false when* $p = 74$*.*

2 Grammars and the Chomsky Hierarchy

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 2.1. 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 [87] and in linguistics, Chomsky [17] 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 2.2. *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 2.3. 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 2.4. 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 2.5. 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 2.6. *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 [17] 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 2.7. *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 regular (or right-linear) 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* $u, v, w \in (N \cup \Sigma)^*$ *and* $v \neq \varepsilon$*; 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 2.8. *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 2.9. 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 2.10. *A language L is generated by a regular expression iff it is generated by a regular 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 $S_1 \rightarrow uS$ without using the rule $S \rightarrow S_1$ iff $S_1 \rightarrow u$ can be derived in the original grammar for L; now one sees that $S \rightarrow^* uS$ for non-empty words in the new grammar is only possible iff $u = u_1u_2 \ldots u_n$ 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 $A \rightarrow w$ 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 $A \rightarrow w$.

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 $A \rightarrow wB$ 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 2.11. 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 2.13, $L_U = \{0,1\}^* \cdot 2$, $L_T = \{0,1\}^* \cdot 2 \cdot L_U = \{0,1\}^* \cdot 2 \cdot \{0,1\}^* \cdot 2$, $L_V = \{0,2\}^*$, $L_W = \{1,2\}^*$ and $L = L_S = L_T \cup L_V \cup L_W$.

Exercise 2.12. *Let L be the language $(\{00,11,22\} \cdot \{33\}^*)^*$. Make a regular grammar generating the language.*

Example 2.13. 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 2.14. *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 2.15: 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) will be proven later using derivation trees of words and Chomsky Normal Form of the grammar; the proof will be given when these tools are introduced. ▮

In Section 3 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 2.16. *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 2.17. *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 2.16 but does not satisfy Theorem 2.15 (a).*

Exercise 2.18. *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 2.15 (a) holds with the constant k being h^2.*

Example 2.19. 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 2.15 (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 2.15 (b). This contradiction proves that L cannot be context-free.

Example 2.20. The language L of all words which have as many 0 as 1 satisfies the pumping condition in Corollary 2.16 but not the pumping condition in Theorem 2.15 (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 2.16 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 2.15 (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 2.15 (a) is not satisfied.

Theorem 2.21. *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 2.15 (a) for regular languages;*
(d) *L satisfies the Theorem 2.15 (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 $h - 1 = \ell \cdot k!/|wy|$ for some integer ℓ, the word vw^hxy^hz is equal to $0^n \cdot 0^{k!\cdot\ell}$. 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 2.22. *Consider the following languages:*

- $L = \{0^n1^n2^n : n \in \mathbb{N}\}$*;*
- $H = \{0^n1^m : n^2 \leq m \leq 2n^2\}$*;*
- $K = \{0^n1^m2^k : n \cdot m = k\}$*.*

Show that these languages are not context-free using Theorem 2.15 (b).

Exercise 2.23. *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 2.15 (a) and 2.15 (b) and Corollary 2.16) are satisfied by L, H *and* K, *respectively.*

Exercise 2.24. *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 2.15 (a) and 2.15 (b) and Corollary 2.16) are satisfied by* L.

Exercise 2.25. *Consider the grammar* $(\{S\}, \{0,1,2,3\}, \{S \to 00S|S1|S2|3\}, S)$ *and construct for the language* L *generated by the grammar the following: a regular grammar for* L *and a regular expression for* L.

In the following exercises, let $f_L(n)$ be the number of words $w \in L$ with $|w| < n$. So if $L = \{0\}^*$ then $f_L(n) = n$ and if $L = \{0,1\}^*$ then $f_L(n) = 2^n - 1$.

Exercise 2.26. *Is there a context-free language* L *with* $f_L(n) = \lfloor \sqrt{n} \rfloor$, *where* $\lfloor \sqrt{n} \rfloor$ *is the largest integer bounded by* $\sqrt{n}$? *Either prove that there is no such set or construct a set with the corresponding context-free grammar.*

Exercise 2.27. *Is there a regular set* L *with* $f_L(n) = n(n+1)/2$? *Either prove that there is no such set or construct a set with the corresponding regular grammar or regular expression.*

Exercise 2.28. *Is there a context-sensitive set* L *with* $f_L(n) = n^n$, *where* $0^0 = 0$? *Either prove that there is no such set or construct a set with the corresponding grammar.*

Exercise 2.29. *Is there a regular set* L *with* $f_L(n) = (3^n - 1)/2 + \lfloor n/2 \rfloor$? *Either prove that there is no such set or construct a set with the corresponding regular grammar or regular expression.*

Exercise 2.30. *Is there a regular set* L *with* $f_L(n) = \lfloor n/3 \rfloor + \lfloor n/2 \rfloor$? *Either prove that there is no such set or construct a set with the corresponding regular grammar or regular expression.*

For the following exercises, call y a pump of $xyz \in L$ iff $|y| \geq 1$ and $\{x\} \cdot \{y\}^* \cdot \{z\} \subseteq L$. For infinite languages L the optimal pump length k as witnessed by h is the smallest number k for which there is a h such that all $w \in L$ with $|w| \geq h$ have a pump of length up to k.

Exercise 2.31. *Determine the optimal pump length k and the witness length h for the language $\{000, 111, 222\}^* \cap \{0000, 1111, 2222\}^* \cap \{00000, 11111, 22222\}^*$ and explain the solution.*

Exercise 2.32. *Determine the optimal pump length k and the witness length h for the language of all words where the length modulo 10 is in $\{1, 3, 7, 9\}$ and explain the solution.*

Exercise 2.33. *Determine the optimal pump length k and the witness length h for the language of all words where the length modulo 10 is in $\{0, 2, 4, 5, 6, 8\}$ and explain the solution.*

Exercise 2.34. *Determine the optimal pump length k and the witness length h for the language $\{001100110011\} \cdot \{222\}^* \cup \{0011\} \cdot \{2222\}^* \cup \{001100110011001100110011\}$ and explain the solution.*

Exercise 2.35. *Determine the optimal pump length k and the witness length h for the language of all decimal numbers without leading zeroes which are multiples of 512 and explain the solution.*

In the following exercises, the task is the following: Given a set H which is interpreted as a set of decimal numbers, find an infinite set $L \subseteq H$ having the property stated in the exercise. Furthermore, if possible L should be regular and a regular expression or grammar should witness this; if L cannot be taken to be regular, but can be taken to be context-free then a context-free grammar should witness that L is context-free and one should use the pumping lemma to show that L cannot be taken regular; if L is context-sensitive then a grammar should witness this and one should use the context-free pumping lemma to prove that no infinite context-free L can solve the task.

Exercise 2.36. *Find infinite $L \subseteq H$ for $H = \{10^n 20^m 1 : n \geq m \geq 1$ and $n + m$ is even$\}$ such that all members of L are square numbers.*

Exercise 2.37. *Find infinite $L \subseteq H$ for $H = \{10^n 30^m 30^k 1 : 2n \geq m + k$ and 3 divides $n + m + k\}$ such that all members of L are third powers (cubes).*

Exercise 2.38. *Find infinite $L \subseteq H$ for $H = \{1\} \cdot \{0\}^+ \cdot \{3\} \cdot \{0\}^+ \cdot \{3\} \cdot \{0\}^+ \cdot \{1\} \cdot \{0\}^+$ such that all members of L are third powers (cubes).*

3 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 3.1: 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 $s = 0$.

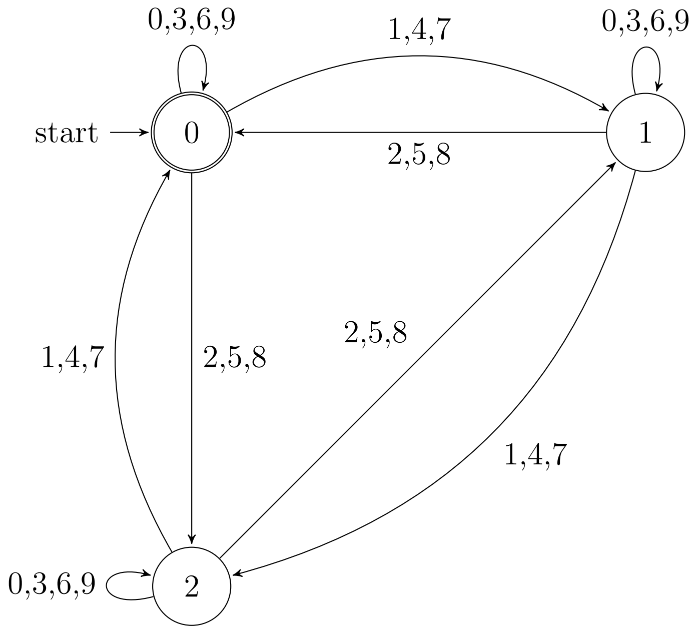
Case s=2 : If $a_m \in \{0, 3, 6, 9\}$ then update $s = 2$;
if $a_m \in \{1, 4, 7\}$ then update $s = 0$;
if $a_m \in \{2, 5, 8\}$ then update $s = 1$.

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

Description 3.3: 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 $\delta : Q \times \Sigma \rightarrow Q$ 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 3.4. 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 3.5. 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.

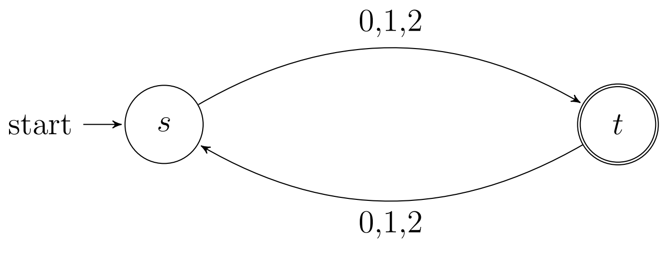
On input ε the algorithm rejects after the first test whether the input is exhausted. On input 00 the algorithm would read b one time and then do the line after the test whether b is 0; as the input is not yet exhausted, the algorithm rejects. On input 0 the algorithm goes the same way until but finally 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 3.6. *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 3.7. *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 3.8: 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 regular 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 4.13.

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 3.9: 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 3.10. 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 3.9 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 $u_0(u_1)^*u_2u_3 \subseteq L$.
- u_2 contains an even number of 0s. Then $u_0u_1(u_2)^*u_3 \subseteq L$.
- u_1 and u_2 contain both an odd number of 0s. Then u_1u_2 contains an even number of 0s and $u_0(u_1u_2)^*u_3 \subseteq L$.

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 $0^i0^{(j-i)h}0^{k-j}1^{k+k!} = 0^{k+k!}1^{k+k!} \notin H$, hence the pumping condition is not satisfied.

Theorem 3.11: Ehrenfeucht, Parikh and Rozenberg [25]. *A language L is regular if and only if both L and its complement satisfy the block pumping lemma.*

However, there are non-regular languages L which satisfy the block pumping lemma.

Quiz 3.12. *Which of the following languages over $\Sigma = \{0, 1, 2, 3\}$ satisfy the pumping-condition from Theorem 3.9:*
(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) $\{w : w$ *contains more* 1 *than* $0\}$*?*

Exercise 3.13. *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 3.14. *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 3.15. *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 3.16. *Give an example L of a language which satisfies Theorem 2.15 (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 3.9 (the Block Pumping Lemma).*

Theorem 3.17: Myhill and Nerode's Minimal DFA [67]. *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 $F = \{L_x : x \in \Sigma^* \wedge \varepsilon \in L_x\}$.

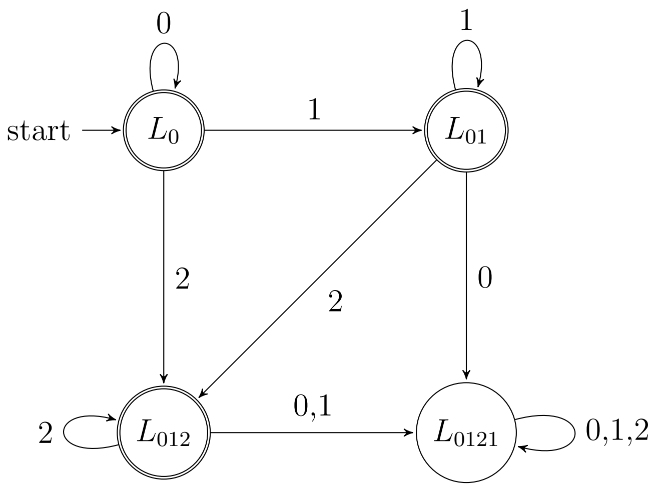
In practice, one would of course pick representatives for each state, so there is a finite subset Q of Σ^* with $\varepsilon \in Q$ and for each set L_y there is exactly one $x \in Q$ with $L_x = L_y$. Then $\delta(x,a)$ is that unique y with $L_y = L_{xa}$.

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 4.2 below).

Remark 3.18. Although the above theorem is published by Anil Nerode [67], 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 [8].

Example 3.19. 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 3.20. 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 3.21: Jaffe's Matching Pumping Lemma [45]. 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 3.22. *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 3.23. *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 3.24. *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 3.24, 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 3.25. *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 3.24 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 3.26. *Is the following statement true: If L satisfies Corollary 3.24 and H is regular then $L \cdot H$ satisfies Corollary 3.24?*

Exercise 3.27. *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 2.15 (a) also satisfy Corollary 3.24? Here x^{mi} is the mirror image of x, so $01122^{mi} = 22110$ and $L^{mi} = \{x^{mi} : x \in L\}$. Prove the answer.*

Exercise 3.28. *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.*

Are the languages L and H regular? If so, provide a dfa. Which of the pumping lemmas (except for the block pumping lemma) do they satisfy?

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 3.29: Myhill's and Nerodes Algorithm to Minimise Deterministic Finite Automata [67].
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 $\delta(q, a)$

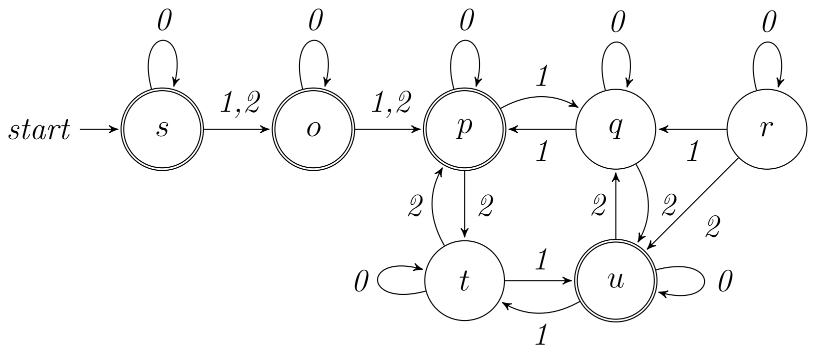
is added to the set R then q is reachable, by some word x; it follows that $\delta(q,a)$ is reachable by the word xa. Furthermore, the adding of nodes is repeated until the set R is closed, that is, for all $q \in R$ and $a \in \Sigma$ the state $\delta(q,a)$ is also in R. Thus one cannot reach from 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 $(\delta(p,a_1a_2\ldots a_m),\delta(q,a_1a_2\ldots a_m))$ goes eventually into γ: by induction hypothesis $(\delta(p,a_1a_2\ldots a_m a_{m+1}),\delta(q,a_1a_2\ldots a_m a_{m+1}))$ is at some point of time going into γ and therefore the pair $(\delta(p,a_1a_2\ldots a_m),\delta(q,a_1a_2\ldots a_m))$ satisfies that, when applying the symbol a_{m+1} to the two states, the resulting pair is in γ, hence $(\delta(p,a_1a_2\ldots a_m),\delta(q,a_1a_2\ldots a_m))$ 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 3.30. *Let the following deterministic finite automaton be given:*



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

Exercise 3.31. *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 3.32. *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 3.33. *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 3.34. *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.

Exercise 3.35. *Provide a regular expression for the language from Exercise 3.33.*

Exercise 3.36. *Provide a regular expression for the language from Exercise 3.34.*

For the following exercises, the task is to use any of the styles above for deterministic automata (graphs, tables as in Example 3.4 and programs as in Example 3.5). Note that for the programs, each variable can only hold one of finitely many predefined values, please specify the range. Furthermore, the programs read the symbols one by one and only remember what is stored at the variables about them (plus the current position in the program). There are commands to read the next symbol and to test whether the input is exhausted.

Exercise 3.37. *Provide a dfa (either as table or program or graph) of the set of all decimal numbers where between between two occurences of a digit d are at least three other digits.*

Exercise 3.38. *Provide a dfa (either as table or program or graph) of the set of all decimal numbers which are not multiples of a one-digit prime number.*

Exercise 3.39. *Provide a dfa (either as table or program or graph) of the set of all decimal numbers with at least five decimal digits which are divisible by* 8.

Exercise 3.40. *Provide a dfa (either as table or program or graph) of the set of all decimal numbers which have in their decimal representation twenty consecutive odd digits.*

Exercise 3.41. *Provide a dfa (either as table or program or graph) of the set of all octal numbers (digits* $0, 1, 2, 3, 4, 5, 6, 7$*) without leading zeroes which are not multiples of* 7.

Exercise 3.42. *Consider the automaton* $(\{0, 1, 2, 3\}, \{0, 1, 2, 3\}, \delta, 0, \{1, 3\})$ *with* δ *given in this table.*

q	*type*	$\delta(q, a)$ *for* $a = 0$	1	2	3
0	*start, rej*	0	1	2	3
1	*acc*	1	1	2	3
2	*rej*	2	2	2	3
3	*acc*	3	3	3	3

Make a regular expression for the language L recognised by the dfa.

Exercise 3.43. *Let L as in Exercise 3.42 and make a regular expression for the language of words of odd lengths in L.*

Exercise 3.44. *Let L as in Exercise 3.42 and make a regular expression for the language of words of length at least 5 in L.*

4 Nondeterministic Finite Automata

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, nondeterminism 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 4.1. 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 4.2: Nondeterministic Finite Automaton. A nondeterministic automaton can guess information and, in the case that it guessed right, verify that a word is accepting.

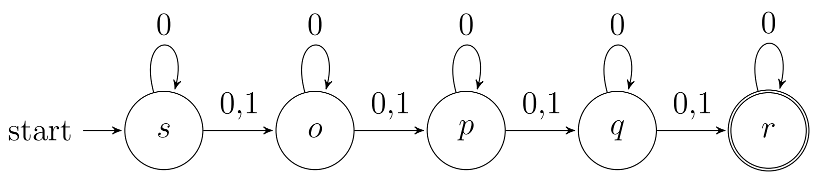
A nondeterministic automaton $(Q, \Sigma, \delta, s, F)$ differs from the deterministic automaton in the way that δ is a multi-valued function, that is, for each $q \in Q$ and $a \in \Sigma$ the value $\delta(q, a)$ is a set of states.

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 nondeterministic 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;

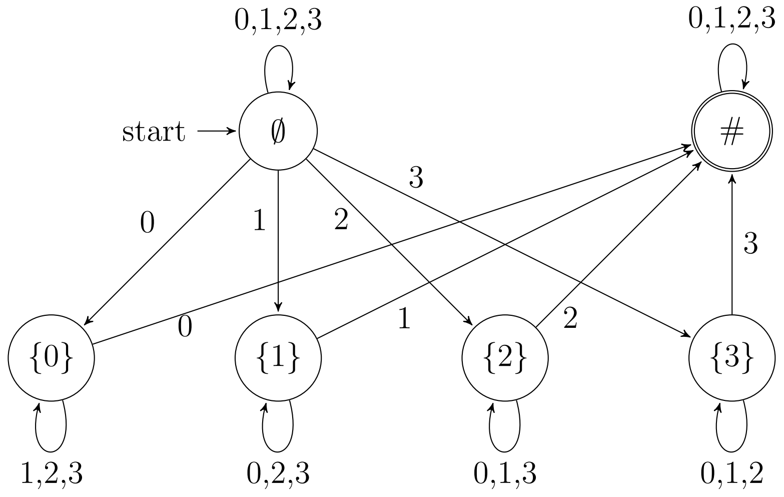
other rejecting runs might also exist. For rejecting a word, all runs which exist must be rejecting, this includes the case that there is no run at all (neither an accepting nor a rejecting).

Example 4.3. Consider the following nondeterministic 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 4.4: Large DFA and small NFA. For the dfa with $2^n + 1$ states from Example 4.1, 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 [9, 10] as well as Rabin and Scott [72]. 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 4.5: Determinisation of NFAs [9, 10, 72]**.** *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 4.6. 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

$\delta'(\emptyset, a) = \emptyset$ for $a \in \{0,1\}$,
$\delta'(\{s\}, 0) = \{s,q\}$, $\delta'(\{s\}, 1) = \{s\}$,
$\delta'(\{q\}, a) = \emptyset$ for $a \in \{0,1\}$,
$\delta'(\{s,q\}, 0) = \{s,q\}$, $\delta'(\{s,q\}, 1) = \{s\}$.

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 $\Sigma = \{0,1\}$.

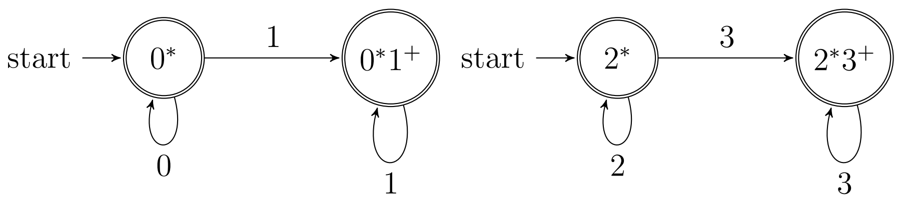
Exercise 4.7. *Consider the language $\{0,1\}^* \cdot 0 \cdot \{0,1\}^{n-1}$:*
(a) *Show that a dfa recognising it needs at least 2^n states;*
(b) *Make an nfa recognising it with at most $n+1$ states;*
(c) *Made a dfa recognising it with exactly 2^n states.*

Exercise 4.8. *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.*

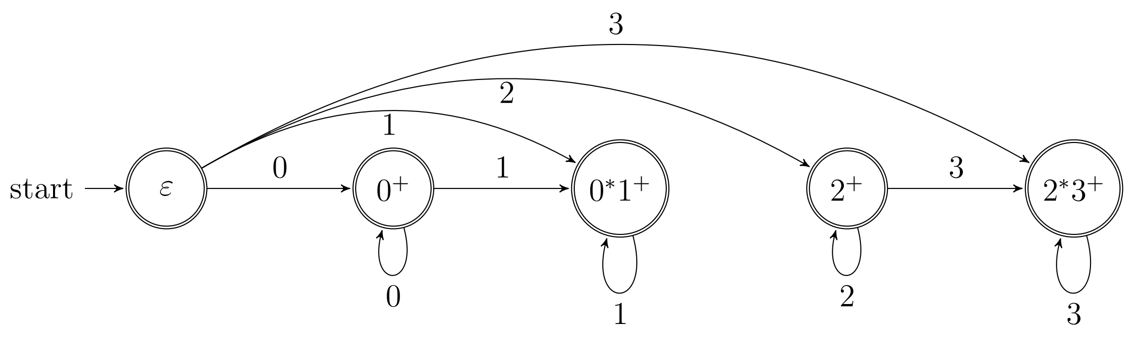
Example 4.9. One can generalise the nfa to a machine $(Q, \Sigma, \delta, I, F)$ where a set I of starting states replaces the single starting state s. Now such a machine accepts a string $w = a_1 a_2 \ldots a_i \in \Sigma^i$ iff there is a sequence $q_0 q_1 \ldots q_i$ of states such that

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

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 4.10. *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 4.9 with $|Q| = n$ which recognises L and that every complete dfa recognising L needs 2^n states.*

The exact trade-off between the numbers of states of an nfa and of a complete dfa was determined by Meyer and Fischer [65]. Their construction does not need the above multiple start states.

Exercise 4.11. *Given an nfa $(\{q_0, q_1, \ldots, q_{n-1}\}, \{0, 1\}, \delta, q_0, \{q_0\})$ with $\delta(q_m, 1) = \{q_{(m+1) \, mod \, n}\}$, $\delta(q_0, 0) = \emptyset$ and $\delta(q_m, 0) = \{q_0, q_m\}$ for $m \in \{1, 2, \ldots, n-1\}$. Determine the number of states of an equivalent complete and minimal dfa and explain how this number is derived.*

Exercise 4.12. *Assume that the alphabet is unary, that is, $\Sigma = \{0\}$. Now show that every nfa with two states over this alphabet has an equivalent dfa with up to three states. For this, carry out the Büchi construction and show that at least one state is not reached.*

Theorem 4.13. *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 $A \to a_1 C_1$, $C_1 \to a_2 C_2$, $\ldots$, $C_{n-1} \to a_n B$. 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 4.14. 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 nondeterministic 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 4.15. *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 nondeterministic finite automaton recognising the language generated by this grammar.*

Exercise 4.16. *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 nondeterministic 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 2.10 showed that a language L is generated by a regular expression iff it has a regular grammar; Theorem 3.8 showed that if L is recognised by a dfa then it L is also generated by a regular expression; Theorem 4.5 showed that if L is recognised by an nfa then L is recognised by a dfa; Theorem 4.13 showed if L is generated by a regular grammar then L is recognised by an nfa. Thus these four concepts are all equivalent.

Corollary 4.17. *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 nondeterministic 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 nondeterministic automata in the sense that a nondeterministic 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 [34] 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 4.18. 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 $x = a_0a_1 \ldots a_{n-1}$ then $10^m \in L_x$ iff $x10^m \in L$ iff $a_0a_1 \ldots a_{n-1}10^m \in L$ iff $a_m = 1$. Thus for $x = a_0a_1 \ldots a_{n-1}$ and $y = b_0b_1 \ldots b_{n-1}$, 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 4.19. *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. ▮

Example 4.20: Inductive Definition of Shortest Word. A counterpart to structural induction are inductive definitions, which can also run along the structure of regular expressions. For this, recall that for an alphabet Σ, the length-lexicographic order chooses the shorter string, if two strings v, w compared are not of the same length, and the lexicographically first string in the case that both strings have the same length. So, for $\Sigma = \{0, 1\}$, the order is $\varepsilon <_{ll} 0 <_{ll} 1 <_{ll} 00 <_{ll} 01 <_{ll} 10 <_{ll} 11 <_{ll} 000 <_{ll} \ldots$ and one uses this length-lexicographical order $<_{ll}$ to define the shortest word of a regular experssion $sw(reg\,exp)$. The inductive definition over the structure of regular expressions follows the following case-distinction:

$$\begin{array}{rcl}
sw(\emptyset) & = & \infty; \\
sw(\{w_1, \ldots, w_n\}) & = & min_{ll}\{w_1, \ldots, w_n\}; \\
sw(\sigma \cup \tau) & = & \begin{cases} sw(\sigma) & \text{if } sw(\tau) = \infty; \\ sw(\tau) & \text{if } sw(\sigma) = \infty; \\ min_{ll}\{sw(\sigma), sw(\tau)\} & \text{otherwise;} \end{cases} \\
sw(\sigma \cdot \tau) & = & \begin{cases} \infty & \text{if } sw(\sigma) = \infty \\ & \text{or } sw(\tau) = \infty; \\ sw(\sigma) \cdot sw(\tau) & \text{otherwise;} \end{cases} \\
sw(\sigma^*) & = & \varepsilon.
\end{array}$$

Now one could also use this structural definition to prove along the above cases that $|sw(\sigma)| \leq |\sigma|$ where $\{, \}, (,), \emptyset, \cup, \cdot, ^*, \infty$ are extra symbols not in Σ which are used in either regular expressions or the output to denote that the regular expression does not produce a word. Furthermore, $|\varepsilon| = 0$. In listings of finite sets, one denotes the empty string by just making a string of length 0 over Σ, for example the input $\{, 00, 001\}$ to sw stands for $\{\varepsilon, 00, 001\}$. Note that $\{\}$ is therefore $\{\varepsilon\}$ and not $\emptyset$. It is left to the reader to adjust the above definition and the treatment of regular expressions such that brackets are taking correctly into account.

Exercise 4.21. *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 4.22. *Is Exercise 4.21 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 4.23: Ehrenfeucht and Zeiger's Exponential Gap [27]**.** 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

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

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

Exercise 4.24. *Assume that an nfa of k states recognises a language L. Show that the language does then satisfy the Block Pumping Lemma (Theorem 3.9) 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 4.25. *Given numbers n, m with $n > m > 2$, provide an example of a regular language where the Block pumping constant is exactly m and where every nfa needs at least n states.*

In the following five exercises, one should try to find small nfas; however, full marks are also awarded if the nfa is small but not the smallest possible.

Exercise 4.26. *Consider the language $H = \{vawa : v, w \in \Sigma^*, a \in \Sigma\}$. Let n be the size of the alphabet Σ and assume $n \geq 2$. Determine the size of the smallest dfa of H in dependence of n and give a good upper bound for the size of the nfa. Explain the results and construct the automata for $\Sigma = \{0, 1\}$.*

Exercise 4.27. *Consider the language $I = \{ua : u \in (\Sigma - \{a\})^*, a \in \Sigma\}$. Let n be the size of the alphabet Σ and assume $n \geq 2$. Determine the size of the smallest dfa of I in dependence of n and give a good upper bound for the size of the nfa. Explain the results and construct the automata for $\Sigma = \{0, 1\}$.*

Exercise 4.28. *Consider the language $J = \{abuc : a, b \in \Sigma, u \in \Sigma^*, c \in \{a, b\}\}$. Let n be the size of the alphabet Σ and assume $n \geq 2$. Determine the size of the smallest dfa of J in dependence of n and give a good upper bound for the size of the nfa. Explain the results and construct the automata for $\Sigma = \{0, 1\}$.*

Exercise 4.29. *Consider the language $K = \{avbwc : a, b \in \Sigma, v, w \in \Sigma^*, c \notin \{a, b\}\}$. Let n be the size of the alphabet Σ and assume $n \geq 2$. Determine the size of the smallest dfa of K in dependence of n and give a good upper bound for the size of the nfa. Explain the results and construct the automata for $\Sigma = \{0, 1\}$.*

Exercise 4.30. *Consider the language $L = \{w : \exists a, b \in \Sigma\, [w \in \{a, b\}^*]\}$. Let n be the size of the alphabet Σ and assume $n \geq 2$. Determine the size of the smallest dfa of L in dependence of n and give a good upper bound for the size of the nfa. Explain the results and construct the automata for $\Sigma = \{0, 1, 2\}$.*

The next exercises deal with Jaffe's Pumping Lemma and its constants.

Exercise 4.31. *Show that an nfa for the language $\{0000000\}^* \cup \{00000000\}^*$ needs only 16 states while the constant for Jaffe's pumping lemma is 56.*

Exercise 4.32. *Generalise the idea of Exercise 4.31 to show that there is a family L_n of languages such that an nfa for L_n can be constructed with $O(n^3)$ states while Jaffe's pumping lemma needs a constant of at least 2^n. Provide the family of the L_n and explain why it satisfies the corresponding bounds.*

Exercise 4.33. *Determine the constant of Jaffe's pumping lemma and the sizes of minimal nfa and dfa for $(\{00\} \cdot \{00000\}) \cup (\{00\}^* \cap \{000\}^*)$.*

5 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.

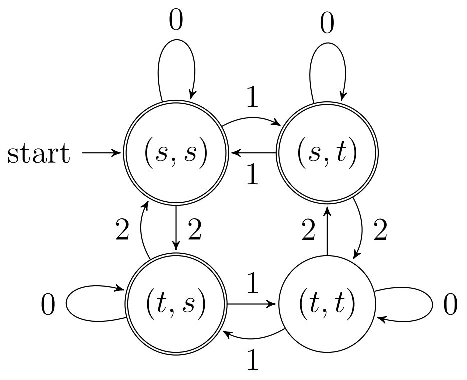
Theorem 5.1: Basic Closure Properties. *Assume that L, H are languages which are on the level CHk of the Chomsky hierarchy. Then the following languages are also on the level CHk: $L \cup H$, $L \cdot H$ and L^*.*

Description 5.2: 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:

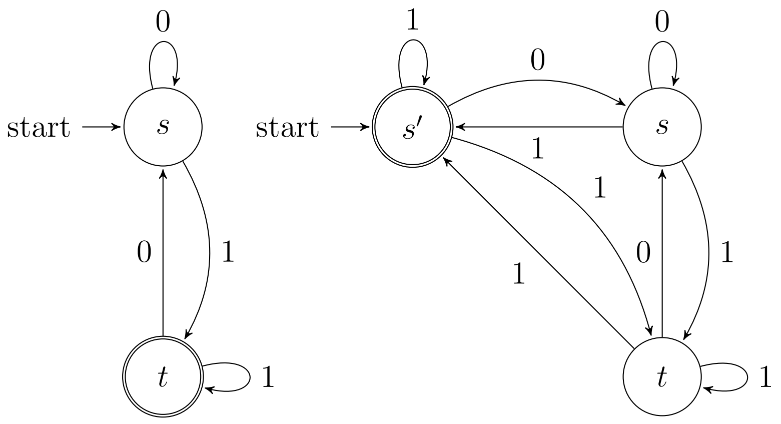
- Union: $F = F_1 \times Q_2 \cup Q_1 \times F_2$;
- Intersection: $F = F_1 \times F_2 = F_1 \times Q_2 \cap Q_1 \times F_2$;
- Difference: $F = F_1 \times (Q_2 - F_2)$;
- Symmetric Difference: $F = F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2$.

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.

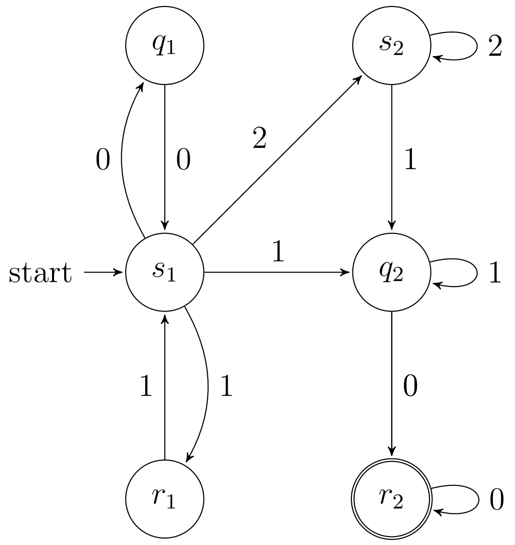
So assume $(Q, \Sigma, \delta, s, F)$ is an nfa recognising L. Now L^* is recognised by $(Q \cup \{s'\}, \Sigma, \delta', s', \{s'\})$ where $\delta' = \delta \cup \{(s', a, p) : (s, a, p) \in \delta\} \cup \{(p, a, s) : (p, a, q) \in \delta$ for some $q \in F\} \cup \{(s', a, s') : a \in L\}$. The last 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 5.3. *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 5.4. *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 5.5. *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 5.6. *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 5.7. *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 5.8. 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 2.22 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 5.9. *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 $L_A = \{w \in \Sigma^* : A \Rightarrow^* w\}$. 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 5.10. *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 5.11. *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 5.12. *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 $S \Rightarrow S_1 S_2 \Rightarrow^* v S_2 \Rightarrow^* vw$. 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 5.13. *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 $S' \to S_1 S_2 S' \mid S_1 S_2 \mid S_1$ 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 $w_1 w_2 \ldots w_{2n}$ by first generating the string $(S_1 S_2)^n$ using the rule $S \to S'$, $n-1$ times the rule $S' \to S_1 S_2 S'$ and one time the rule $S' \to S_1 S_2$. 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 5.14. *Recall that the language* $L = \{0^n 1^n 2^n : n \in \mathbb{N}\}$ *is context-sensitive. Construct a context-sensitive grammar for* L^*.

Theorem 5.15. *The intersection of two context-sensitive languages is context-sensitive.*

Proof Sketch. Let (N_k, Σ, P_k, S) be grammars for L_1 and L_2. Now make a new non-terminal set $N = (N_1 \cup \Sigma \cup \{\#\}) \times (N_2 \cup \Sigma \cup \{\#\})$ with start symbol $\binom{S}{S}$ and following types of rules:
(a) Rules to generate and manage space;
(b) Rules to generate a word v in the upper row;
(c) Rules to generate a word w in the lower row;
(d) Rules to convert a string from N into v provided that the upper components and lower components of the string are both v.

(a): $\binom{S}{S} \to \binom{S}{S}\binom{\#}{\#}$ for producing space; $\binom{A}{B}\binom{\#}{C} \to \binom{\#}{B}\binom{A}{C}$ and $\binom{A}{C}\binom{B}{\#} \to \binom{A}{\#}\binom{B}{C}$ for space management.

(b) and (c): For each rule in P_1, for example, for $AB \to CDE \in P_1$, and all symbols $F, G, H, \ldots$ in N_2, one has the corresponding rule $\binom{A}{F}\binom{B}{G}\binom{\#}{H} \to \binom{C}{F}\binom{D}{G}\binom{E}{H}$. So rules in P_1 are simulated in the upper half and rules in P_2 are simulated in the lower half and they use up # if the left side is shorter than the right one.

(d): Each rule $\binom{a}{a} \to a$ for $a \in \Sigma$ is there to convert a matching pair $\binom{a}{a}$ from $\Sigma \times \Sigma$ (a nonterminal) to a (a terminal).

The idea of the derivation of a word w is then to first use rules of type (a) to produce a string of the form $\binom{S}{S}\binom{\#}{\#}^{|w|-1}$ and afterwards to use the rules of type (b) to derive the word w in the upper row and the rules of type (c) to derive the word w in the lower row; these rules are used in combination with rules for moving # to the front in the upper or lower half. If both derivations have produced terminal words in the upper and lower half (terminals in the original grammar, not with respect to the new intersection grammar) and if these words match, then one can use the rules of type (d) which are $\binom{a}{a} \to a$ for terminals a to indeed derive w. However, if the derivations of the words in the upper row and lower row do not match, then the rules of type (d) cannot derive any terminal word, as there are symbols of the type $\binom{a}{b}$ for different terminals a, b in the original grammar. Thus only words in the intersection can be derived this way. If ε is in the derivation, some special rule can be added to derive ε directly from a new start state which can only be mapped to either ε or $\binom{S}{S}$ by a derivation rule. ∎

Example 5.16. *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. Nonmatching 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. ▮

Exercise 5.17. *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 2.15 (a) and 2.15 (b), Corollary 2.16 and Theorem 3.9 does the language satisfy? Determine its exact position in the Chomsky hierarchy.*

Exercise 5.18. *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 5.19: 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." This palindrome was from Leigh Mercer (1893-1977), a British hobby-writer, who created lots of palindromes, Eckler [23] lists at the end of his article 100 of them.

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 5.20. *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 5.21. *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 5.22. *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 5.23. *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.*

In the following, one considers regular expressions consisting of the symbol L of the language of palindromes over $\{0,1,2\}$ and the mentioned operations. What is the most difficult level in the hierarchy "regular, linear, context-free, context-sensitive" such expressions can generate. It can be used that the language $\{10^i10^j10^k1 : i \neq j, i \neq k, j \neq k\}$ is not context-free.

Exercise 5.24. *Determine the maximum possible complexity of the languages given by expressions containing L and $\cup$ and finite sets.*

Exercise 5.25. *Determine the maximum possible complexity of the languages given by expressions containing L and $\cup$ and $\cdot$ and Kleene star and finite sets.*

Exercise 5.26. *Determine the maximum possible complexity of the languages given by expressions containing L and $\cup$ and $\cdot$ and $\cap$ and Kleene star and finite sets.*

Exercise 5.27. *Determine the maximum possible complexity of the languages given by expressions containing L and $\cdot$ and set difference and Kleene star and finite sets.*

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

Definition 5.28: 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 5.29. 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 5.30. *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 5.31. *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 5.32. *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 $l \to r$ 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 5.33. 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 5.34. *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 5.33 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 5.35. *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 5.36. *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

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

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

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 5.37: Illustrating Theorem 5.36. 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 5.38. *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 $\ell_0 = 0$, ℓ_1,..., ℓ_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 5.39. *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 5.40. *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 5.41. *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* $1s$ *than it has* $0s\}$.

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 5.42. *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 5.43. *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 5.44. *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. This system was used by Floyd and Knuth [29] to carry out operations on register machines (which can as unit operations add, subtract and compare natural numbers) in linear time.

Exercise 5.45. *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 5.46: 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 5.47. *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 5.48. *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 $h^{-1}(L)$.*

Description 5.49: Generalised Homomorphism. A generalised homomorphism is a mapping h from regular subsets of Σ^* to regular subsets of Γ^* for some alphabets Σ, Γ is a generalised homomorphism iff it preserves $\emptyset$, union, concatenation and Kleene star. That is, h must satisfy for all regular subsets H, L of Σ^* the following conditions:

- $h(\emptyset) = \emptyset$;
- $h(L \cup H) = h(L) \cup h(H)$;
- $h(L \cdot H) = h(L) \cdot h(H)$;
- $h(L^*) = (h(L))^*$.

Note that $\emptyset^* = \{\varepsilon\}$ and therefore $h(\{\varepsilon\}) = h(\emptyset^*) = (h(\emptyset))^* = \emptyset^* = \{\varepsilon\}$. Furthermore, for words v, w, $h(\{vw\}) = h(\{v\}) \cdot h(\{w\})$ which implies that one knows $h(\{u\})$ for all words u whenever one knows $h(\{a\})$ for all symbols a in the alphabet.

Examples 5.50. First, the mapping $L \mapsto L \cap \{\varepsilon\}$ is a generalised homomorphism. Second, if one maps the empty set to $\emptyset$ and every regular nonempty subset of Σ^* to $\{\varepsilon\}$, this is also a generalised homomorphism and would work for every target alphabet Γ. Third, the identity mapping $L \mapsto L$ is a generalised homomorphism from regular subsets of Σ^* to regular subsets of Σ^*.

Exercise 5.51. *Show that whenever $h : \Sigma^* \to \Gamma^*$ is a homomorphism then the mapping $L \mapsto \{h(u) : u \in L\}$ is a generalised homomorphism which maps regular subsets of Σ^* to regular subsets of Γ^*.*

Exercise 5.52. *Let h be any given generalised homomorphism. Show by structural induction that $h(L) = \bigcup_{u \in L} h(u)$ for all regular languages L. Furthermore, show that every mapping h satisfying $h(\{\varepsilon\}) = \{\varepsilon\}$, $h(L) = \bigcup_{u \in L} h(\{u\})$ and $h(L \cdot H) = h(L) \cdot h(H)$ for all regular subsets L, H of Σ^* is a generalised homomorophism. Is the same true if one weakens the condition $h(\{\varepsilon\}) = \{\varepsilon\}$ to $\varepsilon \in h(\{\varepsilon\})$?*

Exercise 5.53. *Construct a mapping which satisfies $h(\emptyset) = \emptyset$, $h(\{\varepsilon\}) = \{\varepsilon\}$, $h(L \cup H) = h(L) \cup h(H)$ and $h(L \cdot H) = h(L) \cdot h(H)$ for all regular languages L, H but which does not satisy $h(L) = \bigcup_{u \in L} h(\{u\})$ for some infinite regular set L.*

Exercise 5.54. *Assume that h is a generalised homomorphism and $k(L) = h(L) \cdot h(L)$. Is k a generalised homomorphism? Prove the answer.*

Exercise 5.55. *Assume that h is a generalised homomorphism and*

$$\ell(L) = \bigcup_{u \in h(L)} \Sigma^{|u|},$$

where $\Sigma^0 = \{\varepsilon\}$. Is ℓ a generalised homomorphism? Prove the answer.

Exercise 5.56. *Let $\Sigma = \{0, 1, 2\}$ and h be the generalised homomorphism given by $h(\{0\}) = \{1, 2\}$, $h(\{1\}) = \{0, 2\}$ and $h(\{2\}) = \{0, 1\}$. Which of the following statements are true for this h and all regular subsets L, H of Σ^*:*

(a) *If $L \neq H$ then $h(L) \neq h(H)$;*
(b) *If $L \subseteq H$ then $h(L) \subseteq h(H)$;*
(c) *If L is finite then $h(L)$ is finite;*

(d) *If L is infinite then $h(L)$ is infinite and has exponential growth.*

Prove the answers. The formula $h(L) = \bigcup_{u \in L} h(\{u\})$ from Exercise 5.52 can be used without proof for this exercise.

Note that one can the following property of the image of regular sets L with respect to a generalised homomorphism h also take as a definition for $h(L)$ in the case that L is not regular:

$$h(L) = \bigcup_{a_1 \ldots a_n \in L} h(a_1) \cdot h(a_2) \cdot \ldots \cdot h(a_n)$$

where the empty concatenation gives ε, so that ε is in $h(L)$ whenever $\varepsilon \in L$. Now one uses the definition in order to construct the image of h for the following three sets:

(a) $I = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}^*$;

(b) $J = \{00, 11, 22\}^* \cdot \{000, 111, 222\}$;

(c) $K = \{0^n 1^n 2^n : n \geq 2\}$.

If h maps 0 to $\{0\}^+$ and $1, 2$ to $\{\varepsilon\}$ then $h(I) = \{0\}^*$, $h(J) = \{\varepsilon\} \cup \{0\}^+ \cdot \{0\}^+$ and $h(K) = \{0\}^+$.

Exercise 5.57. *Determine $h(I)$, $h(J)$ and $h(K)$ for I, J, K as above where h is given by $h(0) = \{3, 4\}^+$, $h(1) = \{3, 5\}^+$ and $h(2) = \{4, 5\}^+$. If possible, provide the languages as regular expressions.*

Exercise 5.58. *Determine $h(I)$, $h(J)$ and $h(K)$ for I, J, K as above where h is given by $h(0) = \{\varepsilon, 3, 33\}$, $h(1) = \{\varepsilon, 4, 44\}$ and $h(2) = \{\varepsilon, 5, 55\}$. If possible, provide the languages as regular expressions.*

Exercise 5.59. *Determine $h(I)$, $h(J)$ and $h(K)$ for I, J, K as above where h is given by $h(a) = \{aaa, aaaa\}^+$ for all letters $a \in \{0, 1, 2\}$. If possible, provide the languages as regular expressions.*

Selftest 5.60. *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 5.61. *Consider the language L from Selftest 5.60. Does this language satisfy the traditional pumping lemma (Theorem 2.15 (a)) for regular languages? If so, what is the optimal constant?*

Selftest 5.62. *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 5.63. *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 5.64. *Assume that an nondeterministic finite automaton has 1000 states and accepts some word. How long is, in the worst case, the shortest word accepted by the automaton?*

Selftest 5.65. *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 5.66. *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 5.67. *Construct a constext-sensitive grammar which accepts a word iff it has the same amount of 0, 1 and 2.*

Selftest 5.68. *Create a context-sensitive grammar for all words of the form $(2w)^k3$ where $k \geq 2$ and w is a binary string.*

Selftest 5.69. *Create a context-sensitive grammar for all words of the form $30^i1^j2^k3$ with $i, j, k \geq 1$ and $i \cdot j = k$ (as a product of natural numbers).*

Solution for Selftest 5.60. The right answer is (a). One can write the language as an extremely long regular expression of the form $\{0,1,2\}^* \cdot v_0v_0 \cdot \{0,1,2\}^* \cup \{0,1,2\}^* \cdot v_1v_1 \cdot \{0,1,2\}^* \cup \ldots \cup \{0,1,2\}^* \cdot v_nv_n \cdot \{0,1,2\}^*$. 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 5.61. For the language L from Selftest 5.60, 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 5.62. 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 5.63. 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:

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

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 5.64. First one considers the nondeterministic 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 5.65. 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 ${}_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^2u_j \ldots u_7$ 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 ${}_0 \ldots u_{i-1}(u_i \ldots u_{j-1})^0u_j \ldots u_7$ 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 5.66. 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 5.67. 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 $\{0, 1, 2\}$.

Solution for Selftest 5.68. $\Sigma = \{0, 1, 2, 3\}$, $N = \{S, T, U, V, W\}$ and the start symbols is S. The rules are $S \to T23$, $T \to T2|U$, $U \to UV|UW|2$, $V0 \to 0V$, $V1 \to 1V$, $V2 \to 02V$, $V3 \to 03$, $W0 \to 0W$, $W1 \to 1W$, $W2 \to 12W$, $W3 \to 13$.

The idea is to create first an U followed by a sequence of 2 and then a 3. The U can send off a V or a W to the right and eventually becomes the first 2. The V moves to the right over 0 and 1; whenever it crosses a 2, it creates a 0 before the 2, when it reaches the 3 at the end, it becomes a 0 before the 3. The W moves to the right over 0 and 1; whenever it crosses a 2, it creates a 1 before the 2, when it reaches the 3 at the end, it becomes a 1 before the 3.

Solution for Selftest 5.69. $\Sigma = \{0, 1, 2, 3\}$, $N = \{S, T, U, V, W\}$ and the start symbols is S. The rules are $S \to TU3$, $T \to TU|V1$, $V \to V1|3$, $1U \to U1W$, $0U \to 00$, $3U \to 30$, $W1 \to 1W$, $WU \to UW$, $W2 \to 2W$, $W3 \to 23$.

The idea is to create a word 31^jU^i3 with the first rules, here $i, j \geq 1$. Now the U can only move to the front until they reach a 3 or a 0, each time they hop over a 1, a W is created. When they reach a 0 or 3, they become a 0. The W move to the back until they reach the back 3 and then they become a 2.

6 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 6.1: 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 6.2: 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 6.3. 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 6.4. *Bring the grammar* $(\{S, T\}, \{0, 1\}, \{S \to TTTT, T \to 0T1|\varepsilon\}, S)$ *into Chomsky Normal Form.*

Exercise 6.5. *Bring the grammar* $(\{S, T\}, \{0, 1\}, \{S \to ST|T, T \to 0T1|01\}, S)$ *into Chomsky Normal Form.*

Exercise 6.6. *Bring the grammar* $(\{S\}, \{0, 1\}, \{S \to 0SS11SS0, 0110\}, S)$ *into Chomsky Normal Form.*

Algorithm 6.7: Removal of Useless Nonterminals. 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 $(N_2, \Sigma, P_2, S)$ does not contain any useless non-terminal.
```

Quiz 6.8. *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 6.9. *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 6.10. *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 6.11: 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 6.2 to make the grammar into Chomsky Normal Form and then Algorithm 6.7 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 6.12: 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 6.11: 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 A 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 6.13. *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 6.14. *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 6.15. *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?*

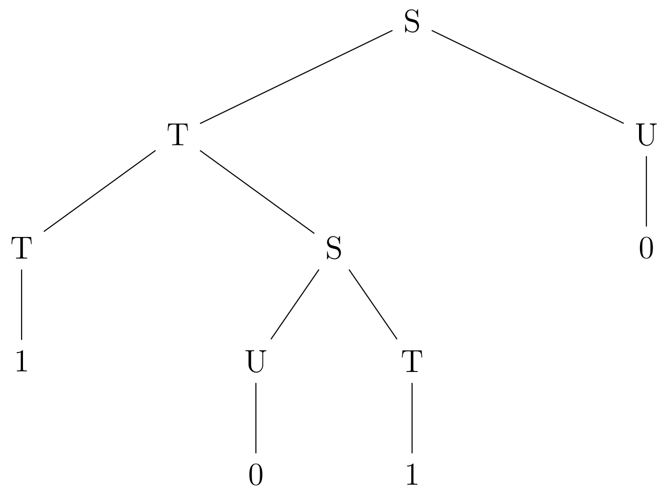
Exercise 6.16. *Consider the grammar* $(\{S, T, U, V, W\}, \{0, 1, 2\}, P, S)$ *with* P *consisting of the rules* $S \to UT|TU|2$, $T \to VV$, $U \to WW$, $V \to 0$, $W \to 1$. *How many words does the grammar generate: (a) None, (b) One, (c) Two, (d) Three, (e) Finitely many and at least four, (f) Infinitely many?*

Exercise 6.17. *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 6.18: 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 6.19. *For the grammar from Description 6.18, how many derivation trees are there to derive* 011001*?*

Exercise 6.20. *For the grammar from Description 6.18, how many derivation trees are there to derive* 000111*?*

Exercise 6.21. *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.

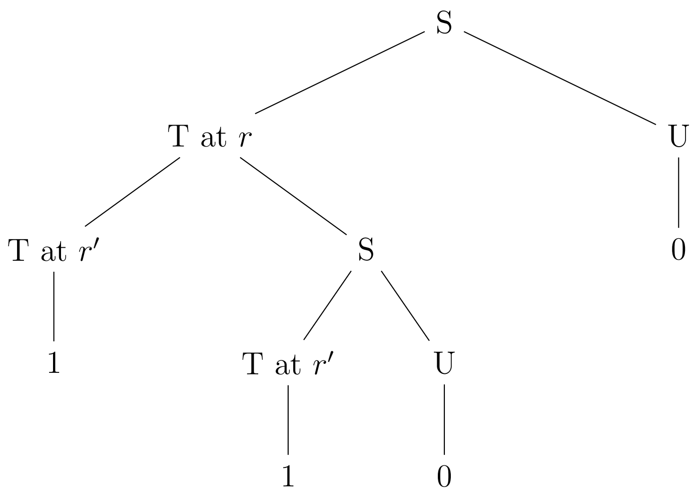
Theorem 2.15 (b). Let $L \subseteq \Sigma^*$ be an infinite context-free language generated by a grammar (N,Σ,P,S) in Chomsky Normal Form with h non-terminals. Then the constant $k = 2^{h+1}$ satisfies that for every $u \in L$ of length at least k there is a representation $vwxyz = u$ such that $|wxy| \leq k$, ($w \neq \varepsilon$ 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 6.18 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 6.22: Ogden's Pumping Lemma [69]. *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 6.23. *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 2.16 with pumps of the length of one symbol; this then implies that also Theorem 2.15 (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 6.24: Square-Containing Words (Ehrenfeucht and Rozenberg [26], Ross and Winklmann [75]). *The language L of the square-containing ternary words is not context-free.*

Proposition 6.25. *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 6.26. *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 2.15 (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 6.27. *Prove that the language*

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

satisfies Theorem 2.15 (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 6.28. *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 6.29. 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 6.30. *Let L be the first language from Example 6.29. Find a grammar in Greibach Normal Form for $L \cap 0^*1^*0^*1^*$.*

Exercise 6.31. *Let H be the second language from Example 6.29. Find a grammar in Greibach Normal Form for $H \cap 0^*1^*0^*1^*$.*

7 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 [33].

Cocke [19], Kasami [52] and Younger [91] 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 7.1: Cocke, Kasami and Younger's Parser [19, 52, 91]**.** 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 7.2. 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 7.3. *Let the grammar be the same as in the previous example. Make the table for the word* 1001.

Exercise 7.4. *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 7.5. *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 7.6: 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 7.7: Parsing of Linear Grammars. Let a linear grammar (N, Σ, P, S) be given in the normal form from Description 7.6 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 7.8. 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 7.9. *For the grammar from Example 7.8, construct the table for the algorithm on the word* 0110110*.*

Exercise 7.10. *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 7.6 and construct then the table of the algorithm for the word 00100*.*

Exercise 7.11. *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 7.12: 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 7.7. 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 7.13. *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 7.7.*

Exercise 7.14. *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 7.7.*

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 7.15. *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 $O(n^3)$.*

Description 7.16: Counting Derivation Trees. One can modify the algorithm of Cocke, Kasami and Younger to count derivation trees. The idea is to replace one entry which says which non-terminals generate a subword by an entry which says how many trees generated from each non-terminal the corresponding subword. Here the trees have the corresponding non-terminal as a root and the symbols of the subwords in the leave and each node with its immediate successors corresponds to a rule in the grammar.

More precisely, given a grammar (N, Σ, P, S) in Chomsky Normal Form and a word $w = a_1 a_2 \ldots a_n$, then define for all positions (i,j) with $1 \leq i \leq j \leq n$ and all $A \in N$ the following notions: $E_{i,j}$ denotes the set of all nonterminals which generate the subword $a_i \ldots a_j$ and $D_{i,j,A}$ denote the number of derivation trees which can, with root A, derive a word $a_i \ldots a_j$; for $A \in N - E_{i,j}$, the corresponding number $D_{i,j,A}$ is 0.

The following recursion-formula defines an algorithm to compute the values of the notions $D_{i,j,A}$: For each $A \in N$ and i, if there is a rule $A \rightarrow i$ in P then $D_{i,i,A} = 1$ else $D_{i,i,A} = 0$. Furthermore, For each $A \in N$ and each i, j with $1 \leq i < j \leq n$, the recrusion formula

$$D_{i,j,A} = \sum_{(B,C):\ A \rightarrow BC\ \in\ P} \ \sum_{k:\ i \leq k < j} D_{i,k,B} \cdot D_{k+1,j,C}$$

holds and it does the following: It sums up for all rules $A \rightarrow BC$ and for all k with $i \leq k < j$ the product $D_{i,k,B} \cdot D_{k+1,j,C}$ which is the product of the the number of trees having root B and generating $a_i \ldots a_k$ and the number of trees having root C and generating $a_{k+1} \ldots a_j$. Thus one uses already these numbers of trees which have been counted before and the fact that every tree with root A which generates a word of at least length 2 can be uniquely decomposed into the rule which maps A to the two successor non-terminals B, C in the left and right successor node of the tree and the subtrees with generate the corresponding words from B and C, respectively; note that the number of leaves of these subtrees determine the value of k.

Once that one has computed these values, one can determine the overall number of derivation trees by just taking the number $D_{1,n,S}$.

Exercise 7.17. *Let P contain the rules $V \rightarrow VV|WW|0$ and $W \rightarrow VW|WV|1$ and consider the grammar $(\{V,W\}, \{0,1\}, P, W)$. How many derivation trees has the word 0011100?*

Exercise 7.18. *Let P contain the rules $V \to VV|WW|0$ and $W \to VW|WV|1$ and consider the grammar $(\{V, W\}, \{0, 1\}, P, W)$. How many derivation trees has the word* 0000111*?*

Exercise 7.19. *Let P contain the rules $U \to VU|UV|2$, $V \to VV|WW|0$ and $W \to VW|WV|1$ and consider the grammar $(\{U, V, W\}, \{0, 1, 2\}, P, U)$. How many derivation trees has the word* 021111*?*

Exercise 7.20. *Let P contain the rules $U \to VU|UV|2$, $V \to VV|WW|0$ and $W \to VW|WV|1$ and consider the grammar $(\{U, V, W\}, \{0, 1, 2\}, P, U)$. How many derivation trees has the word* 010012*?*

For context-sensitive languages, only a quadratic-space algorithm is known due to a construction by Savitch [78]. 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 7.21. 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 7.22. 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 7.23. *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 7.24. *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 7.25. *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 7.26: Growing CTS by Dahlhaus and Warmuth [20]**.** *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 7.27: Growing CTS Membership is in P (Dahlhaus and Warmuth [20]). *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 7.28. 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 $S \Rightarrow T11 \Rightarrow T0U11 \Rightarrow 00U0U11 \Rightarrow 00U01111 \Rightarrow 000UU1111 \Rightarrow 000U111111 \Rightarrow 00011111111$. 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 7.29. *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 7.30. *Modify the proof of Theorem 5.38 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 7.31. *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 7.28.*

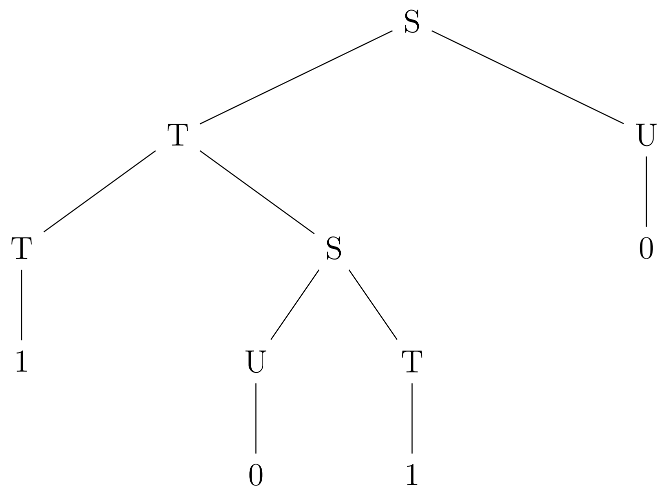
8 Nondeterministic Membership Testing

For finite automata, nondeterministic 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. Nondeterministic 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 8.1: Push Down Automaton. The basic idea for the linear time algorithm to check nondeterministically 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

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

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 nondeterministically 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 8.2: 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 8.3. 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 8.4. 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 8.3.

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

Exercise 8.6. *Construct a pushdown automaton accepting by empty stack for the language* $\{0^n 1^m 2^k : n + m < k\}$.

Exercise 8.7. *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 8.8. 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 8.9. *Construct a pushdown automaton accepting by state for the language* $\{0^n 1^m 2^k : n+m > k\}$.

Exercise 8.10. *Construct a pushdown automaton accepting by state for the language* $\{0^n 1^m 2^k : n \neq k \text{ or } m \neq k\}$.

Exercise 8.11. *Construct a pushdown automaton accepting by state for the language* $\{w \in \{0,1\}^* : w \text{ is not a palindrome}\}$.

Theorem 8.12. *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, $\delta'(q, v, A) = \delta(q, v, A) \cup \{(t, \varepsilon) : v = \varepsilon \text{ and } q \in F\}$;
- 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 8.13: 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 8.14. 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 [37] 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 8.15: 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 $A \in N$,
  $\delta(s, a, S') = \{(t, B_1 B_2 \ldots B'_n) : S \to aB_1 B_2 \ldots B_n$ is a rule in P with $n > 0\} \cup \{(u, \varepsilon) : S \to a$ is a rule in $P\}$;
  $\delta(t, a, A) = \{(t, B_1 B_2 \ldots B_n) : A \to aB_1 B_2 \ldots B_n$ is a rule in P with $n \geq 0\}$;
  $\delta(t, a, A') = \{(t, B_1 B_2 \ldots B'_n) : A \to aB_1 B_2 \ldots B_n$ 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 8.16. *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 8.17. 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 nondeterministic choice. Thus the run of the pushdown automaton is deterministic.

Exercise 8.18. *Provide context-free grammars generating the language of Example 8.17 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 8.19. 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 8.20. *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 8.21. *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 8.22. *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^nUS 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^nTS 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 8.23. *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 8.22 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 8.21; 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 8.24. *Show that the language $L = \{0^n 1 0^m : n \geq m\}$ is deterministic context-free. What about L^*? Give reasons for the answer, though a full proof is not requested.*

Exercise 8.25. *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 8.26. *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 8.27. *Is L^{mi} deterministic context-free whenever L is? Give reasons for the answer, though a full proof is not requested.*

There has been a lot of research under which condition one can in a grammar in Greibach normal form find out by inspecting the next terminals in the word which rule applies. One can consult the Wikipedia pages on LL-grammars and LL(1)-grammars or see the textbook by Waite and Goos [90]. The following exercise investigate the connections between the forms of the derivatives and grammars in Greibach normal form where there is at most one rule of the form $A \to bw$ for each non-terminal A and terminal b.

Exercise 8.28. *Assume that L is recognised by a grammar in Greibach normal form such that for every $b \in \Sigma$ and every nonterminal $A \in N$ there is exactly one rule $A \to bw$ with $w \in N^*$ in the grammar. Show that there is a finite family of languages $H_1, \ldots, H_n$ with $H_1 = L$, $H_2 = \{\varepsilon\}$ and $H_3 = \emptyset$ such that for every $a \in \Sigma$ and every H_k, the derivative $(H_k)_a$ is either an H_ℓ or a product of several of the H_ℓ. Note that $\emptyset_a = \emptyset$ for all $a \in \Sigma$.*

Exercise 8.29. *Assume L is prefix-free and $L \neq \{\varepsilon\}$ and L satisfies that every derivative of L is the product of some fixed languages $H_1, \ldots, H_n$. Is then L is recognised by a grammar in Greibach normal form where for every $A \in N$ and $b \in \Sigma$ there is at most one rule in the grammar of form $A \to bw$?*

Exercise 8.30. *Consider the language L of all ternary words which have as many 0 as 1. Show that every derivative of L is the product of several items of L, L_0 and L_1 but that there is no grammar in Greibach normal form for L which has for every $A \in N$ and $b \in \Sigma$ at most one rule of the form $A \to bw$ with $w \in N^*$.*

Exercise 8.31. *Consider the context-free language L over the alphabet $\{f, (,), 0, 1, ,\}$ with the last symbol being a comma. The rules of the grammar are $S \to f(S,S)|0|1$*

and create all expressions of a binary function f from $\{0,1\}^2$ to $\{0,1\}$. Construct for L a grammar in Greibach normal form where for each pair (A, b) there is at most one rule $A \to bw$ in the grammar.

Exercise 8.32. *Prove the following rules of the derivative with $x \in \Sigma^*$ and $a \in \Sigma$:*

(a) $(L \cup H)_x = L_x \cup H_x$ *and* $(L \cap H)_x = L_x \cap H_x$*;*
(b) *If* $\varepsilon \in L$ *then* $(L \cdot H)_a = L_a \cdot H \cup H_a$ *else* $(L \cdot H)_a = L_a \cdot H$*;*
(c) $(L^*)_a = L_a \cdot L^*$*.*

The following theorem characterises the context-free languages by stating that all derivatives have to be formed from a finite set of languages using concatenation and union; furthermore, all the derivatives of these finitely many languages satisfy the same condition.

Theorem 8.33. *A language L is context-free iff there is a finite list of languages $H_1, H_2, \ldots, H_n$ with $L = H_1$ such that for every word x and every H_m, $(H_m)_x$ is a finite union of finite products of some H_k.*

Exercise 8.34. *Prove this theorem using the rules of Exercise 8.32 and the existence of grammars in Greibach Normal Form for context-free languages.*

Selftest 8.35. 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 8.36. 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 8.37. Consider the language L generated by the grammar $(\{S,T\},\{0,1,2\},\{S \to 0S2|1S2|02|12|T2, T \to 0T|1T\}, S)$. Provide grammars for L in Chomsky Normal Form, in Greibach Normal Form and in the normal form for linear languages.

Selftest 8.38. Carry out the Algorithm of Cocke, Kasami and Younger with the word 0122 with the grammar in Chomsky Normal Form from Selftest 8.37. Provide the table and the decision of the algorithm.

Selftest 8.39. 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 8.37. Provide the table and the decision of the algorithm.

Selftest 8.40. 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 8.41. 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 8.35. 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 8.36. 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 8.37. 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: Nonterminals 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: Nonterminals S, T; Terminals $0, 1, 2$; Rules $S \to 0ST|1ST|0T|1T$, $T \to 2$; start symbol S.

Normal Form of linear grammar: Nonterminals S, T; Terminals $0, 1, 2$; Rules $S \to 0T|1T$, $T \to S2|2$; start symbol S.

Solution for Selftest 8.38.

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

As $S \in E_{1,4}$, the word 0122 is generated by the grammar.

Solution for Selftest 8.39.

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

As $S \in E_{1,4}$, the word 0022 is generated by the grammar.

Solution for Selftest 8.40. 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 8.41. 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 $L \cdot L$.

9 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 9.1: Turing Machine [89]**.** 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 9.2. 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 9.3. *Construct a Turing machine which computes the function $x \mapsto 3x$ where the input x as well as the output are binary numbers.*

Exercise 9.4. *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 $b = 1a_1a_2 \ldots a_n$, 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 9.5: 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.

Register machines of this type were first studied in detail by Hartmanis and Simon [39] and subsequently by Floyd and Knuth [29] who called them "addition machines".

Example 9.6. 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: $R_3 = R_3 + 1$;
Line 7: $R_4 = R_5$;
Line 8: Goto Line 4;
Line 9: Return($R_3$).
```

Exercise 9.7. *Write a program P which computes for input x the value $y = 1 + 2 + 3 + \ldots + x$.*

Exercise 9.8. *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 9.9. *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 9.10: 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 9.11: 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 $(\gamma, q, \mathit{movement})$ is $\gamma \cdot |Q| \cdot 2 + q \cdot 2 + \mathit{movement}$ where $\mathit{movement} = 1$ for going right and $\mathit{movement} = 0$ 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 9.12. *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 9.13. *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 9.14: Turing's Universal Turing Machine [89]**.** 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 9.15. *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 9.16: Primitive Recursive Functions [82]**.** *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 9.17. 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 9.18. *Prove that every function of the form $h(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + b$ with fixed parameters $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$ is primitive recursive.*

Exercise 9.19. *Prove that the function $h(x) = 1 + 2 + \ldots + x$ is primitive recursive.*

Exercies 9.20. *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 [35] 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 9.21: Partial recursive functions [35]. *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 9.22. *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 = $ F($R_2, R_3$);
Line 4: If $R_4 = R_1$ Then Goto Line 8;
Line 5: $R_5 = $ 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: $R_4 = F(R_3, R_1, R_2)$;
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 9.23. *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 [89]; therefore the thesis is also known as "Church–Turing Thesis".

Thesis 9.24: 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 9.25. *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 9.26. *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 9.27. 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($R_4$).
```

Alternatively, one can do in linear time by mimicking the school algorithm for binary numbers. For a bit compacter program, several commands per line are allowed.

```
Line 1: Function Binarymult($R_1, R_2$);
Line 2: $R_3 = 1$; $R_4 = 1$; $R_5 = 0$; $R_6 = R_2$;
Line 3: If $R_3 > R_6$ Then Goto Line 5;
Line 4: $R_3 = R_3 + R_3$; Goto Line 3;
Line 5: $R_6 = R_6 + R_6$; $R_5 = R_5 + R_5$; $R_4 = R_4 + R_4$;
Line 6: If $R_6 < R_3$ Then Goto Line 8;
Line 7: $R_5 = R_5 + R_1$; $R_6 = R_6 - R_3$;
Line 8: If $R_4 < R_3$ Then Goto Line 5;
Line 9: Return($R_5$).
```

In this program, R_6 initially holds the input and later the input times some power of 2. R_3 is doubled until it is larger than R_2 and is a power of 2. In the loop starting in Line 5, R_4 is then used as counter going to R_4, but again by doubling up in order to need the same time. In the loop body, the highest order bit of R_6 is read out and R_5 is updated accordingly. R_5 is doubled in each iteration of the loop in order to accomodate then the processing of lower order bits of R_6.

Exercise 9.28. *Write a register program which computes the remainder in polynomial time.*

Exercise 9.29. *Write a register program which divides in polynomial time.*

Exercise 9.30. *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.*

Floyd and Knuth [29] called such register machines "addition machines" and also showed that they can multiply, divide and form remainders in linear time (with the same primitive steps as here for register machines). Their method used the representation of numbers by Fibonacci numbers and their method is superior to the above. However, for avoiding copy and paste, please use in the solutions to the before exercises one of the methods indicated in Example 9.27.

10 Complexity Considerations

To determine the exact computational complexity of an algorithm or problem, one has to agree on the machine model and the primitive operations permitted. For Turiing machines, for example, there is a big difference induced by the number of tapes which they use (this number is always constant, but depends on the machine model). For example, for a one-tape Turing machine where the input is already on the tape, the only languages they can recognise in linear time are the regular sets.

Theorem 10.1. *A Turing machine with only one tape can recognise in time linear of the input exactly the regular sets.*

The idea of this proof is the following. Trakhtenbrot [88] and Hartmanis [38] have proven that a one-tape Turing machine which runs linear time visits every cell on the Turing tape at most constantly many times. Now one can modify the computation such that one enlarges the alphabet to tuples of symbols and at every visit one adds new components to the tuple which were empty before and one does not overwrite the old nonempty components. So if a cell has been visited twice, its tuple is $(a_0, s_1, a_1, s_2, a_2, 0, \ldots, 0)$ where s_1, s_2 are the states after the first and second visit and a_1, a_2 are the corresponding states when the automaton reaches the cell. So at each subsequent visit, the first two empty components are replaced by state and new symbol; here one assumes that the combination of state and tape symbol always gives away the direction the Turing machine takes as well the information whether it goes into halting-accept or halting-reject. Note that the tuple length is constant, as each cell is visited at most only a constant amount of times. Now the nfa, when processing the input word, guesses these tuples and verifies that they are a consistent and accepting computation of the given word; note that the computations only go beyond both ends a constant number of steps at most, as otherwise the computation would be nonterminating. Thus every language whose characteristic function is computed by a linear time Turing machine is a regular set.

Exercise 10.2. Extend the above proof-sketch to a full proof of the result. This is a more difficult exercise, as one has to reconstruct or look up the above cited result as well, it should be included into the explanations; the exercise goes beyong the material needed for this course.

Exercise 10.3. A Turing machine with two tapes can move the head independently on the two tapes. Show that the set of palindromes can be recgonised by a Turing machine with two tapes in linear time.

Also register machines (in the model of Floyd and Knuth [29] where they can add, subtract and compare integers) have a different computational power than multitape

Turing machines. Here a comparison of these two models. The complexity of the basic operations is given for best known algorithm, for example, while the school book multiplication gives $O(n^2)$ on n-bit numbers, the state of the art is $O(n \log n)$ by Harvey and van der Hoeven [40]; for their algorithm they use the model of a multitape Turing machine.

Complexities	Register Machine Floyd and Knuth [29]	Multitape Turing machine Turing [89]
Addition	1	$\Theta(n)$
Subtraction	1	$\Theta(n)$
Comparison	1	$\Theta(n)$
Multiplication	$\Theta(n)$	$O(n \log n)$
Bitwise And, Or, ...	$\Theta(n)$	$\Theta(n)$
Doubling	1	$\Theta(n)$
Halving	$\Theta(n)$	$\Theta(n)$
Regular Set Membership	$\Theta(n)$	$\Theta(n)$

This table shows the dependence of complexities of the chosen model. One cannot say that one model is superior to the other because the operations like multiplication are faster, instead the primitive operations are just more powerful. Floyd and Knuth [29] were the first to show that the basic operations like multiplying, dividing and computing remainders are $O(n)$ in their model; actually they cannot be faster. Stockmeyer [83] showed that even a register machine with adding, subtracting, comparing and multiplying as primitive operations, it needs $\Omega(n)$ operations to compute $x/2$ for an n-bit number x; similarly to determine whether the number x is even or odd takes $\Omega(n)$ operations; thus with the upper bounds by Floyd and Knuty [29], the linear bounds for various operations are tight. Regular set membership means that if one looks at the sequence of bits in the binary representation, then the question is whether the so obtained word is in a given regular set. For the multitape Turing machine model, lower bounds $\Omega(n)$ stem already from the trivial fact that for various operations, the Turing machine has to access all bits in order to carry out the operation correctly.

Description 10.4: The Complexity Class P. As the complexity of a computation depends on the exact way the process of computing is formalised, one tries to come up with robust classes which are the same for many formalisation. One of them is the class **P** of sets which can be decided by an algorithm in polynomial time. The main idea is that there is a polynomial p (bound of form $k \cdot (n+k)^k$ for some $k > 0$) and the computing device can use $p(n)$ steps to compute the answer for n-bit input x. In some cases, one does not count the number of bits, but, in particular for finite graphs, the number n of nodes or for strings over Σ just the number of symbols (without converting the alphabet into words over binary alphabet).

Suppose that two computation models C and D are given. Now the following conditions guarantee that C cannot compute more in polynomial time than D iff there is a polynomial p and a constant c such that the following holds:

1. Every primitive operation of C on inputs of length n can be done by $p(n)$ steps of D;
2. Every primitive operation of C with inputs up to length n produces only outputs up to length $n + c$.

If now an algorithm needs $q(n)$ steps with respect to C for some input of length n then all intermediate values have length up to $n + c \cdot q(n)$ and need $q(p(n + c \cdot q(n)))$ steps which is, as three polynomials are concatenated, again a polynomial bound on the number of steps in the computation according to model D.

Example 10.5. The model of Floyd and Knuth [29] satisfies both conditions, when compared to multitape Turing machines.

First, multitape Turing machines can add and subtract and compate n-bit numbers in $O(n)$ steps, thus the first condition holds.

Second, if x, y are numbers with up to n bits then $x + y$ has at most $n + 1$ bits, as the absolute value of $x + y$ is at most the double of the maximum of the absolute values of x and y. Comparisons produce only one bit of output. Furthermore, adding or subtracting constants will make short inputs longer by more than one bit, but the maximum of the bits gained are given by the length of the longest constant in the program. Thus the second condition is satisfied.

Indeed, Hartmanis and Simon [39] showed that register machines with the primitive operation to add, to subtract, to compare and to carry out bitwise operations can compute in polynomially many steps the same what Turing machines can compute. On the other hand, they showed if one adds to this instruction set also the multiplication, classes considered to be larger like **NP** and **PSPACE** can be handled with polynomially many steps. So one can characterise **P** with multitape Turing machines or with register machines which can add, subtract and compare plus, perhaps, bitwise operations.

Description 10.6. The Complexity Class **NP** Nondeterminism is quite common in automata theory; nfas can compute with nondetermism not more than dfas, but they can be much smaller. For transducers, nondeterminism can be essential and certain functions can only be computed with nondeterministic transducers. Similarly one can introduce nondeterminism for Turing machines or register machines. Machines can do nondeterministically a branching (either to go this state / line or to that state / line).

Furthermore, for register machines one might guess the content of a register; however, again one needs a limitation, so if x is the old content of the register, then the new one is an element of $\{0, 1, 2, \ldots, x-1, x\}$ and nothing else, as guessing extremely large numbers might allow to go beyond **NP**. Now a set X is in the class **NP** iff there is a nondeterministic register machine and a polynomial p such that for each input y of length n, the nondeterministic register machine runs at most $p(n)$ steps and $y \in X$ iff there is an accepting run of the register machine.

Example 10.7. Stockmeyer [83] showed that deterministic register machines need $\Omega(n)$ steps to check whether an input number is even or odd. The following *nondeterminnistic program* does it in constantly many steps.

```
Line 1: Function Remainderbytwo($R_1$);
Line 2: Guess $R_2$ from range $\{0, 1, \ldots, R_1\}$;
Line 3: $R_3 = R_2 + R_2$;
Line 4: If $R_3 = R_1$ Then Return 0;
Line 5: $R_3 = R_3 + 1$;
Line 6: If $R_3 = R_1$ Then Return 1;
Line 7: Reject Computation.
```

The program terminates with correct output iff the value R_2 guessed satisfies $2 \cdot R_2 \leq R_1 \leq 2 \cdot R_2 + 1$. All other guesses lead to a rejection of the computation. In other words, the register machine has to guess the downrounded half of R_1.

Description 10.8: NP-Complete Problems. A problem A is **NP**-complete iff for every further problem B in **NP** there is a polynomial time computable function f such that $x \in B$ iff $f(x) \in A$.

Example 10.9. Manders and Adleman [61] provided the following numerical problem which is **NP**-complete: Given three natural numbers a, b, c, are there two natural numbers x, y with $a \cdot x^2 + b \cdot y = c$.

Example 10.10: The Satisfiability Problem SAT. Let $x_1, x_2, \ldots, x_n$ be $\{0, 1\}$-valued variables, for example bits of an n-bit number. A clause is a condition of the form $x_i = b_i \vee x_j = b_j \vee x_k = b_k$ where b_i, b_j, b_k are bits. Let F be a set of clauses of size m. Now the instance F has a solution iff there are values for the variables $x_1, \ldots, x_n$ such that all clauses in F are satisfied.

SAT is now the set of all (n, F) such that F consists of clauses over the first n variables and F can be solved. **3SAT** is the set of all $(n, F) \in$ **SAT** where each clause in F consists of at most three conditions; analogously one defines **2SAT** and **4SAT**

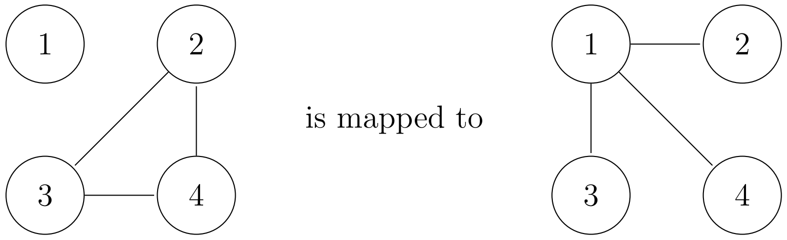
and so on. The problem **2SAT** is in **P** while the problems **3SAT**, **4SAT**, ... and **SAT** itself are all **NP**-complete.

Example 10.11: The Graph Problem CLIQUE. Let (V, E) be a graph with n nodes and let m with $1 \leq m \leq n$ be a natural number. Now **CLIQUE** is the set of all such (V, E, n, m) where V has n nodes and a subset W of m nodes such that each two distinct nodes in W are connected by an edge in E. **CLIQUE** is **NP**-complete.

Example 10.12: The Graph Problem INDEPENDENTSET. Let (V, E) be a graph with n nodes and let m with $1 \leq m \leq n$ be a natural number. Now **INDEPENDENTSET** is the set of all such (V, E, n, m) where V has n nodes and a subset W of m nodes such that no two distinct nodes in W are connected by an edge in E.

Definition 10.13. *A set A is polynomial-time many-one reducible to a set B iff there is a polynomial-time computable function g such that, for all x, $x \in A \Leftrightarrow g(x) \in B$.*

Example 10.14. CLIQUE is polynomial-time many-one reducible to **INDEPENDENTSET** by the following mapping F: Given a graph (V, E) with n nodes and a parameter m, let $F(V, E)$ be the set of all edges between distinct of nodes of V which are not in E then the mapping f which maps (V, E, n, m) to $(V, F(V, E), n, m)$ is a polynomial-time many-one reduction satisfying that (V, E, n, m) is in **CLIQUE** iff $(V, F(V, E), n, m)$ is in **INDEPENDENTSET**. As **INDEPENDENTSET** is in **NP** and as **CLIQUE** is polynomial-time many-one reducible to **INDEPENDENTSET**, the problem **INDEPENDENTSET** is **NP**-complete.



This graphics shows the reduction from $(V, E, 4, 3)$ to $(V, F(V, E), 4, 3)$, the first graph has a clique of size three, the second an independent set of size three. In both cases the witnessing set is $\{2, 3, 4\}$. The graphs are in **CLIQUE** and **INDEPENDENTSET**, respectively. The main idea of the reduction is to swap edges and nonedges.

Description 10.15: Complexity Class PSPACE. The space complexity is the space used for the computations. In the case of a register machine, it can be viewed as the number of bits needed to store the registers, in other words, a space bound of

m bits imposes that all registers used are during the whole runtime of the program strictly between -2^m and 2^m. Now a problem is in **PSPACE** if there is a polynomial p such that one can, for inputs of n bits, solve the problem with computations obeying to a space bound of $p(n)$ bits. Note that **NP** is contained in **PSPACE** as one can just try out all possible solutions and then either to settle with the first one to fit the problem requirements or to find out that there is no solution.

Example 10.16. A **PSPACE**-complete problem is quantified satisfiability **QSAT** where there are variables $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ and where **QSAT** is the set of all instances (F, n, m) of m clauses which satisfy that for all x_1 there is y_1 for all x_2 there is y_2 … for all x_n there is y_n such that all clauses in F are satisfied.

Exercise 10.17. Make a proof that every deterministic register program whose space bound (register size) is bounded by $p(n)$ bits throughout the overall time runs at most in time $2^{O(p(n))}$ for the same polynomial p.

Exercise 10.18. Show that the following problem is in **NP**: **CONNECTED-HALVES** is the set of all graphs (V, E) such that one can split V into two two subsets U, W such that $|U| \leq |W| \leq |U| + 1$ and every node in U is connected to every node in W by an edge.

While the complexity classes above are more at the lower end of the hierarchy of complxity classes, the following will be on the upper end.

Description 10.19: Primitive Recursive. 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: $R_3$ = 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: $R_1 =$ Mult($R_1, 3$) $+ 1$;
Line 5: Goto Line 2;
Line 6: $R_1 =$ 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 $R_1 =$ 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 $R_1 =$ 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 10.20. *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 10.21. *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 10.22. *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 10.22, one can make many equivalent formalisations for the notion that a set is enumerated by an algorithm.

Description 10.23: Decidable Sets. The class of all sets which has a recursive decision procedure is called the class **DECIDABLE** of decidable (or recursive) sets. It is the class of all sets of numbers for which some register machine can compute the characteristic function. One can make a program F in two inputs e, x such that the restriction $x \mapsto F(e, x)$ is the characteristic function of the e-th primitive recursive set; this is possible as one can enumerate all computer programs which satisfy the above specifications of primite recursive programs. The so obtained set is decidable but not primitive recursive. When it comes to functions, the above mentioned Ackermann function is recursive but not primitive recursive.

Theorem 10.24. *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 $A = \{x : \exists y\, [g(x, y) > 0]\}$.*

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 10.22, 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 10.22 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 10.25. *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 10.26. *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 10.27: Halting Problem [89]**.** 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 10.28: Undecidability of the Halting Problem [89]**.** *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 [89] 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 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 10.24 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. ▮

In summary, this chapter investigated the hierarchy

$$\mathbf{P} \Rightarrow \mathbf{NP} \Rightarrow \mathbf{PSPACE} \Rightarrow \mathbf{PRIMREC} \Rightarrow \mathbf{DECIDABLE} \Rightarrow \mathbf{RE}$$

and except for the first two, all implications are known to be proper. The problem whether **P** = **NP** is one of the seven Millenium Problems and not solved so far. The last result of this chapter and the subsequent exercises study properties of recursively enumerable and recursive sets.

Theorem 10.29. *A set L is recursive iff both L and $\mathbb{N}-L$ are recursively enumerable.*

Exercise 10.30. *Prove this characterisation.*

Exercise 10.31. *Prove that the set $\{e : \varphi_e(2e+5)$ is defined$\}$ is undecidable.*

Exercise 10.32. *Prove that the set $\{e : \varphi_e(e^2+1)$ is defined$\}$ is undecidable.*

Exercise 10.33. *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 10.34. *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 10.35. *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 10.36. *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.*

Exercise 10.37. *Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the range of a function which can be computed in polynomial time.*

Exercise 10.38. *Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the domain of a function f where the graph $\{(x, f(x)) : x \in dom(f)\}$ can be decided in polynomial time, that is, given inputs x, y, one can decide in polynomial time whether $(x, y) = (x, f(x))$.*

Exercise 10.39. *Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the domain of a $\{0,1\}$-valued function f where the graph $\{(x, f(x)) : x \in dom(f)\}$ can be decided in polynomial time.*

11 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 11.1. *$A \subseteq \mathbb{N}$ 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 11.2. *The conditions* **(a)** *through* **(d)** *in Definition 11.1 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 11.3. 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 11.4. *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 11.5. *Show that the set of all natural numbers which are multiples of* 5 *but not multiples of* 7 *is Diophantine.*

Exercise 11.6. *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 11.7. *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 11.8. *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 11.9. *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 [59, 60] 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 11.10: Unsolvability of Hilbert's Tenth Problem [59]. *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 11.11. 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 11.12. 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 11.13: Configuration and Update of a Register Machine [89]**.** 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 11.14: 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 11.15: 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 [18] and Turing [89] 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 11.16. *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 11.17: Acceptable Numbering [35]**.** 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 11.18. *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 11.19: Rice's Characterisation of Index Sets [74]. *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 11.18, 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 11.20. *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 11.21. *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 11.22. *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 11.23. 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 11.24. 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 11.25. *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 11.26. *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 11.27. *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 11.28. *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 11.29. *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 11.30. *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 11.31. *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.

12 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 12.1: 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 12.2. *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 12.3. *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 12.4. *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 12.5. *Provide a translation for a conditional jump: If* $R_1 \leq R_2$ *then Goto Line 200. The values of* R_1, R_2 *after doing the conditional jump should be the same as before the translation of the command.*

Corollary 12.6. *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 12.7. *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 5.39 that $h(L \cap H)$ is generated by some grammar. ∎

Exercise 12.8. *In the format of the proof before and with respect to the sample multi counter machine from Definition 12.1, give the encoded version (as word from* $\{0,1,2,3,4\}^+$*) of the run of the machine on the input* 001.

Exercise 12.9. *In the format of the proof before and with respect to the sample multi counter machine from Definition 12.1, give the encoded version (as word from* $\{0,1,2,3,4\}^+$*) of the run of the machine on the input* 001111000.

Theorem 12.10. *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 12.11. *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 12.7 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 12.12: 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

$$x_{k_1} x_{k_2} \ldots x_{k_m} = y_{k_1} y_{k_2} \ldots y_{k_m},$$

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 12.13. *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 12.14. *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 12.15. *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 12.16: Nondeterministic machines. Nondeterminism 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 nondeterminism 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. Nondeterministic 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 nondeterministic 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 nondeterminism to characterise the regular and context-sensitive languages via Turing machines or register machines.

Theorem 12.17. *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 nondeterministic 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 "nondeterministic" can be replaced by "deterministic", as it can be done for finite automata.

Theorem 12.18. *A language L is regular iff there is a nondeterministic 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 "nondeterministic Turing machine" by "deterministic Turing machine"; however, the result is stated here in the more general form.

Example 12.19. 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 12.20. *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 12.18. Determine a regular expression for this language.*

Exercise 12.21. *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 12.22. *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 12.23. 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 12.24. 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 12.25. Construct a context-sensitive grammar for all words in $\{0\}^+$ which have length 2^n for some n.

Selftest 12.26. 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 12.27. Determine, in dependence of the number of states of a nondeterministic 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 12.28. 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 12.29. 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 12.30. 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 12.31. 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 12.32. 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 12.33. Write a register machine program which computes the function $x \mapsto x^8$. All macros used must be defined as well.

Selftest 12.34. 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 12.35. 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 12.36. Is it decidable to test whether a context-free grammar generates infinitely many elements of $\{0\}^* \cdot \{1\}^*$?

Solution for Selftest 12.23. 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 12.24. 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 12.25. 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 12.26. 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 12.27. Assume that L is recognised by a nondeterministic 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 $q_0q_1 \ldots q_n$ 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 $q_0q_1 \ldots q_iq_{j+1} \ldots q_n$ is a run of the automaton on xz and if y is repeated, one can repeat the sequence from q_itoq_j accordingly. So $q_0 \ldots q_i(q_{i+1} \ldots q_j)^3q_{j+1} \ldots q_n$ is an accepting run on xy^3z. Thus all words in xy^*z are accepted by the nondeterministic 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 12.28. 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 12.29. 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 12.30. 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 12.31. 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:

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

As $S \in E_{1,4}$, the word 0100 is in the language.

Solution for Selftest 12.32. 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 12.33. 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 12.34. 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 12.35. 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 12.36. 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.

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] Achim Blumensath and Erich Grädel. Automatic structures. *15th Annual IEEE Symposium on Logic in Computer Science*, LICS 2000, pages 51–62, 2000.

[6] 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.

[7] 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.

[8] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the Association of Computing Machinery*, 11:481–494, 1964.

[9] 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.

[10] 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.

[11] Cristian Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li and Frank Stephan. Deciding parity games in quasipolynomial time. *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, Montreal, QC, Canada, June 19-23, 2017. Pages 252–263, ACM, 2017.

[12] 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.

[13] John Case, Sanjay Jain, Samuel Seah and Frank Stephan. Automatic functions, linear time and learning. *Logical Methods in Computer Science*, 9(3), 2013.

[14] Christopher Chak, Rūsiņš Freivalds, Frank Stephan and Henrietta Tan. On block pumpable languages. *Theoretical Computer Science*, 609:272–285, 2016.

[15] Ashok K. Chandra, Dexter C. Kozen and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[16] 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`.

[17] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[18] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[19] John Cocke and Jacob T. Schwartz. *Programming languages and their compilers: Preliminary notes.* Technical Report, Courant Institute of Mathematical Sciences, New York University, 1970.

[20] Elias Dahlhaus and Manfred K. Warmuth. Membership for Growing Context-Sensitive Grammars Is Polynomial. *Journal of Computer and System Sciences*, 33:456–472, 1986.

[21] Christian Delhommé. Automaticité des ordinaux et des graphes homogènes. *Comptes Rendus Mathematique*, 339(1):5–10, 2004.

[22] Rodney G. Downey and Michael R. Fellows. *Parameterised Complexity.* Springer, Heidelberg, 1999.

[23] A. Ross Eckler. Leigh Mercer, Palindromist. *Word Ways*, 24(3):131–138, 1991.

[24] Lawrence C. Eggan. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal*, 10(4):385–397, 1963.

[25] Andrzej Ehrenfeucht, Rohit Parikh and Grzegorz Rozenberg. Pumping lemmas for regular sets. SIAM Journal on Computing, 10:536–541, 1981.

[26] Andrzej Ehrenfeucht and Grzegorz Rozenberg. On the separating power of EOL systems. *RAIRO Informatique théorique* 17(1): 13–22, 1983.

[27] Andrzej Ehrenfeucht and H. Paul Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[28] David Epstein, James Cannon, Derek Holt, Silvio Levy, Michael Paterson and William Thurston. *Word Processing in Groups*. Jones and Bartlett Publishers, Boston, Massachusetts, 1992.

[29] Robert W. Floyd and Donald E. Knuth. Addition machines. *SIAM Journal on Computing*, 19(2):329–340, 1990.

[30] Péter Frankl. An extremal problem for two families of sets. *European Journal of Combinatorics* 3:125–127, 1982.

[31] Dennis Fung. Automata Theory: The XM Problem. BComp Dissertation (Final Year Project), School of Computing, National University of Singapore, 2014.

[32] 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.

[33] 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`

[34] Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. *ACM Transactions in Computational Logic*, 13(1):4, 2012.

[35] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatshefte für Mathematik und Physik*, 38: 173–198, 1931.

[36] Mark Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[37] 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.

[38] Juris Hartmanis. Computational complexity of one-tape Turing machine computations. *Journal of the Association of Computing Machinery* 15:411–418, 1968.

[39] Juris Hartmanis and Janos Simon. On the power of multiplication in random access machines. *Fifteenth Annual Symposium on Switching and Automata Theory*, SWAT 1974, pages 13–23, IEEE, 1974.

[40] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, to appear. https://hal.archives-ouvertes.fr/hal-02070778v2/document.

[41] Bernard R. Hodgson. *Théories décidables par automate fini.* Ph.D. thesis, University of Montréal, 1976.

[42] Bernard R. Hodgson. Décidabilité par automate fini. *Annales des sciences mathématiques du Québec*, 7(1):39–57, 1983.

[43] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Third Edition, Addison-Wesley Publishing, Reading Massachusetts, 2007.

[44] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[45] Jeffrey Jaffe. A necessary and sufficient pumping lemma for regular languages. *ACM SIGACT News*, 10(2):48–49, 1978.

[46] 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.

[47] 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.

[48] 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.

[49] Sanjay Jain, Yuh Shin Ong and Frank Stephan. Regular patterns, regular languages and context-free languages. *Information Processing Letters* 110:1114–1119, 2010.

[50] 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.

[51] Marcin Jurdziński, Mike Paterson and Uri Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

[52] Tadao Kasami. *An efficient recognition and syntax-analysis algorithm for context-free languages.* Technical Report, Air Force Cambridge Research Laboratories, 1965.

[53] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. *Logical and Computational Complexity*, (International Workshop LCC 1994). Springer LNCS 960:367–392, 1995.

[54] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and its Applications.* Birkhäuser, 2001.

[55] 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.

[56] Lars Kristiansen and Juvenal Murwanashyaka. Decidable and undecidable fragments of first-order concatenation theory. *Fourteenth Conference on Computability in Europe*, CiE 2018, Kiel, Germany, *Springer LNCS* 10936:244-253, 2018. See also https://arxiv.org/abs/1804.06367.

[57] 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.

[58] 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.

[59] 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.

[60] Yuri Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, Massachusetts, 1993.

[61] Kenneth Manders and Leonard Adleman. NP-complete decision problems for quadratic polynomials. *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC 1976, pages 23–29, 1976.

[62] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[63] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[64] George H. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, 34(5):1045–1079, 1955.

[65] Albert R. Meyer and Michael J. Fischer. Economy of description by automata, grammars, and formal systems. *Twelfth Annual Symposium on Switching and Automata Theory*, SWAT 1971, pages 188–191, 1971.

[66] 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.

[67] Anil Nerode. Linear automaton transformations. *Proceedings of the AMS*, 9:541–544, 1958.

[68] Maurice Nivat. Transductions des langages de Chomsky. *Annales de l'institut Fourier*, Grenoble, 18:339–455, 1968.

[69] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2:191–194, 1968.

[70] Viktor Petersson and Sergei G. Vorobyov. A randomized subexponential algorithm for parity games. *Nordic Journal of Computing*, 8:324–345, 2001.

[71] Jean-Éric Pin. On two combinatorial problems arising from automata theory. *Annals of Discrete Mathematics*, 17:535–548, 1983.

[72] Michael O. Rabin and Dana Scott. Finite Automata and their Decision Problems, *IBM Journal of Research and Development*, 3:115–125, 1959.

[73] Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

[74] Henry Gordon Rice. Classes of enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74:358–366, 1953.

[75] Rockford Ross and Karl Winklmann. Repetitive strings are not context-free. *RAIRO Informatique théorique* 16(3):191–199, 1982.

[76] Shmuel Safra. On the complexity of ω-automata. Proceedings twenty-ninth IEEE Symposium on Foundations of Computer Science, pages 319-327, 1988.

[77] Shmuel Safra. Exponential determinization for omega-Automata with a strong fairness acceptance condition. *SIAM Journal on Computing*, 36(3):803–814, 2006.

[78] Walter Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[79] Sven Schewe. Solving parity games in big steps. *FCTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS 4855:449–460, 2007.

[80] Sven Schewe. Büchi Complementation Made Tight. *Symposium on Theoretical Aspects of Computer Science* (STACS 2009), pages 661-672, 2009.

[81] 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.

[82] 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.

[83] Larry J. Stockmeyer. *Arithmetic versus Boolean operations in idealized register machines*, IBM Thomas J. Watson Research Center report RC 5954, 21 April 1976.

[84] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the European Association for Theoretical Computer Science*, 96–100, 1987.

[85] Wai Yean Tan. *Reducibilities between regular languages.* Master Thesis, Department of Mathematics, National University of Singapore, 2010.

[86] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936.

[87] Axel Thue. Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. *Norske Videnskabers Selskabs Skrifter, I, Mathematisch-Naturwissenschaftliche Klasse* (Kristiania), 10, 34 pages, 1914.

[88] Boris A. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logika*, 3:33-48, 1964.

[89] 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.

[90] William M. Waite and Gerhard Goos. *Compiler Construction.* Texts and Monographs in Computer Science. Springer, Heidelberg, 1984.

[91] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:198–208, 1967.

[92] 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.

[93] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200:135–183, 1998.