Chapter 2
Languages, Decidability, and Complexity

Stefan Haar and Tomáš Masopust

2.1 Introduction

Control problems for discrete-event systems or hybrid systems typically involve manipulation of languages that describe system behaviors. This chapter introduces basic automata and grammar models for generating and analyzing languages of the Chomsky hierarchy, as well as their associated decision problems, which are necessary for the understanding of other parts of this book. Notions of decidability of a problem (that is, is there an algorithm solving the given problem?) and of computational complexity (that is, how many computation steps are necessary to solve the given problem?) are introduced. The basic complexity classes are recalled. This chapter is not intended to replace a course on these topics but merely to provide basic notions that are used further in this book, and to provide references to the literature.

In the following, we introduce the basic terminology, notation, definitions, and results concerning the Chomsky hierarchy of formal languages to present the necessary prerequisites for understanding the topic of this chapter, that is, the devices recognizing and generating languages. As this is only an introductory material, not all details and proofs are presented here. Usually, only the basic ideas or sketches of proofs are presented. Further details can be found in the literature, see, e.g., [4, 5, 10, 12, 13, 15].

Stefan Haar
INRIA/LSV, CNRS & ENS de Cachan, 61, avenue du Président Wilson, 94235 Cachan Cedex, France, e-mail: Stefan.Haar@inria.fr

Tomáš Masopust
Institute of Mathematics, Academy of Sciences of the Czech Republic, Žižkova 22, 616 62 Brno, Czech Republic, e-mail: masopust@math.cas.cz

2.2 Regular Languages and Automata

This section introduces the simplest type of languages and automata. First, however, let us define the fundamental concepts. The cardinality of a set A is denoted by $|A|$. For two sets A and B, we write $A \subseteq B$ to denote that A is a subset of B. If $A \subseteq B$ and $A \neq B$, we write $A \subsetneq B$. The notation 2^A denotes the set of all subsets of A.

2.2.1 Words and Languages

An *alphabet* (also called an *event set*) is a finite, nonempty set Σ of abstract elements, which are called *symbols* or *letters*. Let $\Sigma = \{a_1, a_2, \dots, a_n\}$ be an alphabet. A *word* or *string* over Σ is a (finite or infinite) concatenation $w = a_1a_2a_3\dots$ of letters $a_i \in \Sigma$. For instance, a and $ccbc$ are words over $\{a,b,c\}$. The *empty word* is a word consisting of zero letters, denoted by ε. It holds that $\varepsilon \cdot w = w \cdot \varepsilon = w$, for any word w. The set of all finite words over Σ is denoted by $\Sigma^* \triangleq \{a_1a_2a_3\dots a_n \mid n \in \mathbb{N},\ a_i \in \Sigma\}$. The set of all nonempty words over Σ is denoted by $\Sigma^+ \triangleq \Sigma^* \setminus \{\varepsilon\}$. A set L is a *language* over Σ if $L \subseteq \Sigma^*$. The *length* of a word w is denoted $|w|$, that is, $|a_1a_2\dots a_n| = n$. Let $|w|_a$ denote the number of a's in w. For instance, $|ccbc| = 4$ and $|ccbc|_b = 1$. Write w^R for the *mirror image* (or *reversal*) of w defined so that for $w = a_1a_2a_3\dots a_n$, $w^R = a_na_{n-1}a_{n-2}\dots a_1$; $ccbc^R = cbcc$. Word $u \in \Sigma^*$ is a *prefix* of $v \in \Sigma^*$ if there exists $u' \in \Sigma^*$ such that $v = uu'$. Dually, u is a *suffix* of v if there exists $u' \in \Sigma^*$ such that $v = u'u$. Furthermore, u is an *infix* or *factor* of v if there exist $u', u'' \in \Sigma^*$ such that $v = u'uu''$, and a *sub-word* of v if there exist $u_i, v_i \in \Sigma^*$ such that $v = v_0u_1v_1u_2\dots u_nv_n$ with $u = u_1u_2\dots u_n$.

For two languages $K, L \subseteq \Sigma^*$, we have the set theoretic operations $K \cup L$, $K \cap L$, $K \setminus L$, $K^c = \Sigma^* \setminus K$, etc. Define the *concatenation* of K and L as

$$K \cdot L \triangleq \{u \cdot v \mid u \in K,\ v \in L\}\,.$$

The *powers* of a language L are defined as follows: $L^0 \triangleq \{\varepsilon\}$, $L^{n+1} \triangleq L^n \cdot L = L \cdot L^n$,

$$L^* \triangleq \bigcup_{n \geqslant 0} L^n \quad \text{and} \quad L^+ \triangleq \bigcup_{n > 0} L^n\,.$$

Finally, we have the *quotient languages*

$$K^{-1} \cdot L \triangleq \{v \in \Sigma^* \mid \exists u \in K :\ u \cdot v \in L\} \text{ and } L \cdot K^{-1} \triangleq \{u \in \Sigma^* \mid \exists v \in K :\ u \cdot v \in L\}\,.$$

A *substitution* is a mapping $\sigma : \Sigma^* \to 2^{\Gamma^*}$ such that $\sigma(\varepsilon) = \{\varepsilon\}$ and $\sigma(xy) = \sigma(x)\sigma(y)$, where $x, y \in \Sigma^*$. A *(homo)morphism* is a substitution σ such that $\sigma(a)$ consists of exactly one string, for all $a \in \Sigma$. We write $\sigma(a) = w$ instead of $\sigma(a) = \{w\}$, i.e., $\sigma : \Sigma^* \to \Gamma^*$. A *projection* is a homomorphism $\sigma : \Sigma^* \to \Gamma^*$ with $\Gamma \subseteq \Sigma$ such that for all $a \in \Sigma$, $\sigma(a) \in \{a, \varepsilon\}$.

2.2.2 Regular Languages

The above language operations can be abstracted into a class $RE(\Sigma)$ of *regular expressions* over an alphabet Σ as follows.

1. $\underline{\emptyset}$ and $\underline{a}$, for $a \in \Sigma$, are in $RE(\Sigma)$;
2. if $E, F \in RE(\Sigma)$, then $(E+F)$, $(E \cdot F)$, $(E^*) \in RE(\Sigma)$;
3. nothing else is in $RE(\Sigma)$.

Regular expressions correspond to languages. To show this relation, we define the operation $L : RE(\Sigma) \to 2^{\Sigma^*}$ by $L(\underline{\emptyset}) \triangleq \emptyset$, $L(\underline{a}) \triangleq \{a\}$, for all $a \in \Sigma$, and

$$L(E+F) \triangleq L(E) \cup L(F)\,, \quad L(E \cdot F) \triangleq L(E) \cdot L(F)\,, \quad L(E^*) \triangleq (L(E))^*\,.$$

The class $\mathsf{Reg}(\Sigma^*)$ of *regular (also called rational) languages* over Σ is the smallest class of languages over Σ that *(i)* contains the empty language $\emptyset$ and all singletons $\{a\}$, for $a \in \Sigma$, and *(ii)* is closed under the operations $\cup$, $\cdot$, and $()^*$.

Example 2.1. The regular expression $a+ba^*b$ represents the language $L(a+ba^*b) = \{a\} \cup L(ba^*b) = \{a\} \cup \{ba^ib \mid i \geqslant 0\}$.

2.2.3 Automata

We now define the first model of dynamical systems to *generate* languages through their behaviors.

Deterministic Automata

Deterministic automata are finite-state machines, where the input changes the current state of the system. These machines have only finite memory.

Definition 2.1. A *deterministic finite automaton* (DFA, for short) over alphabet Σ is a quintuple $A = (Q, \Sigma, f, q_0, Q_m)$, where Q is a *finite* set of states, $f : Q \times \Sigma \to Q$ is a *transition function* (total or partial), $q_0 \in Q$ is an *initial state*, and $Q_m \subseteq Q$ is a set of *final or marked states*.

DFAs are associated to languages via the following central notions.

Definition 2.2. For $q_1, q_2 \in Q$ and $a \in \Sigma$, write $q_1 \xrightarrow{a} q_2$ if $f(q_1, a) = q_2$. *A accepts* word $w = a_1 a_2 \ldots a_n \in \Sigma^*$ if there exist $q_i \in Q$ such that

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n \in Q_m;$$

write $q_0 \xrightarrow{w} q_n$. Using this definition, we can extend f to be a function from $Q \times \Sigma^*$ to Q so that $f(q_0, w) = q_n$ if $q_0 \xrightarrow{w} q_n$, where $q \xrightarrow{\varepsilon} q$ for all $q \in Q$. The *language accepted (or recognized) by A* is defined as the set

$$L(A) \triangleq \{w \in \Sigma^* \mid \exists q \in Q_m : \; q_0 \xrightarrow{w} q\}.$$

The class $\mathsf{Rec}(\Sigma^*)$ of *recognizable* languages in Σ is formed by those languages $L \subseteq \Sigma^*$ for which there exists a DFA A over Σ such that $L = L(A)$.

Fig. 2.1 presents a simple example of a DFA which accepts or recognizes a language described by the regular expression $ba^*b + a$.

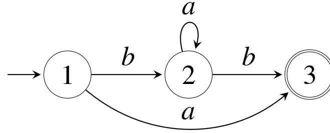

Fig. 2.1 A DFA $A = (\{1,2,3\},\{a,b\},f,1,\{3\})$ which accepts the language ba^*b+a, where f is defined by the arrows.

Non-Deterministic Automata

It is often convenient or natural to work with non-deterministic models in the sense of the following definition, cf. state 2 in Fig. 2.2.

Definition 2.3. A *nondeterministic finite automaton* (NFA, for short) over Σ is a quintuple $A = (Q,\Sigma,T,\mathbf{I},Q_m)$, where Q is a *finite* set of states, $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*, $\mathbf{I} \subseteq Q$ is a set of *initial states*, and $Q_m \subseteq Q$ is a set of *final or marked states*. For $q_1,q_2 \in Q$ and $a \in \Sigma$, write $q_1 \xrightarrow{a} q_2$ if $(q_1,a,q_2) \in T$. *A accepts* word $w = a_1a_2\dots a_n \in \Sigma^*$ if there exist $q_i \in Q$ such that $q_0 \in \mathbf{I}$ and $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \in Q_m$; write $q_0 \xrightarrow{w} q_n$. The *language accepted (or recognized) by A* is defined as the set $L(A) \triangleq \{w \in \Sigma^* \mid \exists q_0 \in \mathbf{I},\; q \in Q_m : \; q_0 \xrightarrow{w} q\}$.

We shall see below that the classes of DFAs and NFAs are equivalent in terms of the recognized languages. First, however, we extend the notion of NFAs so that ε-transitions are allowed. This corresponds to a situation where the automaton changes the state, but reads no input letter.

ε-Automata

Definition 2.4. An NFA $A = (Q,\Sigma,T,\mathbf{I},Q_m)$ such that $T \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is called an ε-automaton, or an NFA with ε-transitions.

Similarly as for the deterministic automata, we can extend T so that $T \subseteq Q \times \Sigma^* \times Q$. Fig. 2.2 presents a simple example of an NFA with ε-transitions which accepts the language $a^*b + a$. The following theorem shows that we can always remove ε-transitions from the automaton.

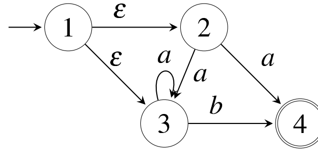

Fig. 2.2 An NFA with ε-transitions that accepts the language a^*b+a.

Theorem 2.1. *For every ε-automaton A there exists an ε-*free *automaton A' such that $L(A) = L(A')$.*

Proof. The proof can be sketched as follows, see Fig. 2.3:

1. For every $q \xrightarrow{\varepsilon} q' \xrightarrow{a} q''$, where $a \in \Sigma$, add $q \xrightarrow{a} q''$;
2. For every $q \xrightarrow{\varepsilon} q'$ with $q' \in Q_m$, add q to Q_m;
3. Remove all ε-transitions. $\square$

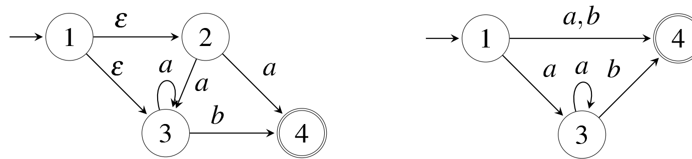

Fig. 2.3 Removing ε-transitions: The NFA of Fig. 2.2 and the ε-free NFA that accepts a^*b+a.

Determinizing NFAs

The disadvantage of the nondeterminism is that we can have two or more choices how to continue from a state. In the right automaton of Fig. 2.3, this situation happens in state 1 for letter a. The following theorem shows that we can always eliminate the nondeterminism by transforming an NFA to a DFA.

Theorem 2.2. *For every NFA (with ε-transitions) $A = (Q, \Sigma, T, \mathbf{I}, Q_m)$ there exists a DFA $A' = (Q', \Sigma, f, q_0, {Q_m}')$ such that $L(A) = L(A')$.*

Proof. The key idea to obtaining the determinized version of A is the following powerset construction.

- $Q' \subseteq 2^Q$;
- initial state q_0 is given by the set of initial states $\mathbf{I}$ *and* those states that are ε-reachable from some initial state: $q_0 \triangleq \{s \in Q \mid \exists s' \in \mathbf{I} : s' \xrightarrow{\varepsilon} s\}$;
- $\forall U \subseteq Q$ and $a \neq \varepsilon : f(U, a) \triangleq \{q \in Q \mid \exists q_u \in U : q_u \xrightarrow{a\varepsilon} q\}$;

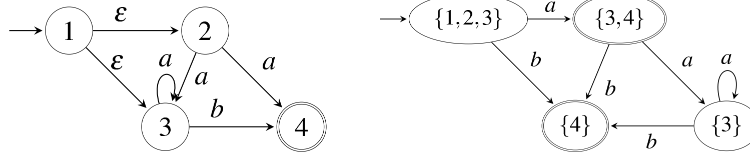

Fig. 2.4 Determinization: The NFA with ε-transitions of Fig. 2.2 and its DFA which accepts the language a^*b+a.

- $Q_m{}' = \{U \subseteq Q \mid U \cap Q_m \neq \emptyset\}$. $\square$

Fig. 2.4 demonstrates the previous theorem. It should be noted that the automaton A' has, in general, a state space whose size is of the order of $2^{|Q|}$, that is, exponential with respect to the state space of A. There exist examples of NFAs for which it can be shown that any language-equivalent DFAs *must* be at least of this size, see [6] and the references therein. Determinization should therefore be used moderately.

2.2.4 Closure Properties of Regular Languages

The algebraic properties considered here for regular languages – and below for other language classes – are important results for testing whether or not a given language belongs to a given class.

Theorem 2.3. $\mathsf{Rec}(\Sigma^*)$ *is closed under the set operations union, intersection, and complement.*

Proof. Let $K, L \in \mathsf{Rec}(\Sigma^*)$ with DFAs A_K, A_L such that $L(A_K) = K$ and $L(A_L) = L$. An NFA to accept $K \cup L$ is obtained by combining A_K and A_L via fusion of initial states. Language $K \cap L$ is accepted by a DFA obtained as a synchronized product of A_K and A_L. In this product automaton, the state set is $Q_K \times Q_L$, and the transition function $f_{prod} : (Q_K \times Q_L) \times \Sigma \to (Q_K \times Q_L)$ is constructed componentwise so that

$$f_{prod}((q_K, q_L), a) = (q'_K, q'_L) \text{ if } f_K(q_K, a) = q'_K \text{ and } f_L(q_L, a) = q'_L .$$

Finally, to obtain a DFA for K^c, it suffices to replace Q_m by $Q \setminus Q_m$ in A_K. $\square$

Theorem 2.4. $\mathsf{Rec}(\Sigma^*)$ *is closed under concatenation, iteration, substitutions, and projection to subalphabets.*

Proof. Let $K, L \in \mathsf{Rec}(\Sigma^*)$ with DFAs A_K, A_L such that $L(A_K) = K$ and $L(A_L) = L$. An automaton $A_{K \cdot L}$ such that $L(A_{K \cdot L}) = K \cdot L$ is obtained by combining A_K and A_L via extra ε-transitions from Q_{mK} to q_{0L}. For A_{L^*}, add to A_L a new state for recognizing ε, plus ε-transitions from Q_{mL} to q_{0L}. If $\sigma : \Sigma \to 2^{\Gamma^*}$ is a substitution, replacing all a-transitions in A_L by a copy of automaton $A_{\sigma(a)}$ yields $A_{\sigma(L)}$. Finally, for any $\Gamma \subseteq \Sigma$ and $a \in \Sigma \setminus \Gamma$, replace $q \xrightarrow{a} q'$ by $q \xrightarrow{\varepsilon} q'$. $\square$

2.2.5 Regularity and Recognizability

The following theorem (Kleene 1936) summarizes the relation between regular expressions and recognizable languages.

Theorem 2.5. $\mathsf{Reg}(\Sigma^*) = \mathsf{Rec}(\Sigma^*)$.

Proof. To prove $\mathsf{Reg}(\Sigma^*) \subseteq \mathsf{Rec}(\Sigma^*)$, we have $\emptyset \in \mathsf{Rec}(\Sigma^*)$ and $\{a\} \in \mathsf{Rec}(\Sigma^*)$, for $a \in \Sigma$ (construct the DFAs). The closure of $\mathsf{Rec}(\Sigma^*)$ under $\cup$, $\cdot$, and $()^*$ then implies $\mathsf{Reg}(\Sigma^*) \subseteq \mathsf{Rec}(\Sigma^*)$. To prove the converse inclusion, $\mathsf{Reg}(\Sigma^*) \supseteq \mathsf{Rec}(\Sigma^*)$, we need to convert DFAs to regular expressions. The idea, depicted in Fig. 2.5, is to construct a regular expression by a step-by-step fusion of transitions. □

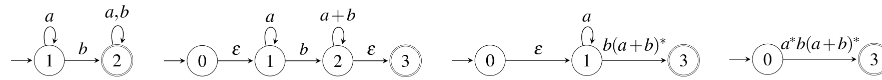

Fig. 2.5 Converting a DFA to a regular expression: The DFA on the left is transformed to the regular expression $a^*b(a+b)^*$.

2.2.6 Criteria for Regularity

We now ask how to identify the limits of regularity. Is it true that all languages are regular? The answer is negative; for instance, the following languages are *not* regular:

- $L_1 = \{a^n b^n \mid n \geqslant 0\}$,
- $L_2 = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$,
- $L_3 = \{w \in \Sigma^* \mid w = w^R\}$.

To prove the non-regularity of these and other languages, a very important tool is the use of results of the following type, which are called *pumping lemmas* or *star/iteration lemmas* in the literature. Of course one may have a choice to apply one or the other result in cases where several such lemmas can be applied.

Lemma 2.1 (Pumping Lemma). *For every $L \in \mathsf{Reg}(\Sigma^*)$ there exists $N \geqslant 0$ such that for all $x \in L$*

- *If $|x| \geqslant N$, then $x = u_1u_2u_3$ with $u_i \in \Sigma^*$, $u_2 \neq \varepsilon$, $|u_1u_2| \leqslant N$, and $u_1u_2^*u_3 \in L$.*
- *If $x = w_1w_2w_3$ with $|w_2| \geqslant N$, then $x = w_1u_1u_2u_3w_3$ with $u_i \in \Sigma^*$, $u_2 \neq \varepsilon$, and $w_1u_1u_2^*u_3w_3 \in L$.*

Proof. The idea for the proofs of this and similar results is as follows. Take a DFA A_L that accepts L. Since the number of states of A_L is finite, every path of length greater than some N that depends on A_L has a loop. Iterating this loop allows to construct the sublanguages whose existence is claimed in Lemma 2.1. □

Example 2.2. To use Lemma 2.1 to show that $L_1 = \{a^n b^n \mid n \geqslant 0\}$ is not regular, assume for contradiction that L_1 is regular. Then there exists N as claimed in Lemma 2.1. Consider the word $w = a^N b^N$. Then, by Lemma 2.1, $w = u_1 u_2 u_3$ with $u_1 u_2^* u_3 \in L_1$. There are three possibilities: $u_2 \in a^*$, $u_2 \in b^*$, or $u_2 \in aa^*bb^*$. In all cases, however, $u_1 u_2 u_2 u_3 \notin L_1$, which is a contradiction.

For proving that a given language L is *non-recognizable*, one often uses a pumping lemma *and* the established closure properties of $\mathsf{Reg}(\Sigma^*)$. For instance, for $L_2 = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$, we have $L_2 \cap a^*b^* = L_1$, hence $L_2 \notin \mathsf{Reg}(\Sigma^*)$.

2.2.7 Minimality

In general, there are infinitely many DFAs and NFAs that accept a given language (for instance, aa^*). We shall see that one can find a *canonical* DFA which is minimal (up to isomorphism) in the number of states. This is not true for NFAs (construct two non-isomorphic two-state automata for the language aa^*).

Definition 2.5. For $u \in \Sigma^*$ and $L \subseteq \Sigma^*$, the *residual language* of L with respect to u is the quotient $u^{-1}L = \{v \in \Sigma^* \mid uv \in L\}$. The *residual automaton* for L is $\mathscr{R}(L) = (Q_L, \Sigma_L, f_L, q_{0L}, Q_{mL})$ such that $Q_L = \{u^{-1}L \mid u \in \Sigma^*\}$, $f_L(u^{-1}L, a) = a^{-1}(u^{-1}L) = (ua)^{-1}L$, $q_{0L} = L = \varepsilon^{-1}L$, $Q_{mL} = \{u^{-1}L \mid \varepsilon \in u^{-1}L\} = \{u^{-1}L \mid u \in L\}$.

Theorem 2.6. *Language $L \subseteq \Sigma^*$ is recognizable if and only if it has* finitely *many residuals. Moreover, any DFA A with $L(A) = L$ allows to inject $\mathscr{R}(L)$ into A by a morphism (*a fortiori*, $\mathscr{R}(L)$ has minimal size among those automata that accept L).*

However, for the construction, a more convenient algorithm is depicted in the following example.

Example 2.3. Consider a slightly modified automaton of Fig. 2.4. In the first step, we add the dead state, d, to make the automaton complete, i.e., to complete the transition function of the automaton. This is done by replacing all the transitions of the form $f(q,a) = \emptyset$ with $f(q,a) = d$. Then, in Step 1, we distinguish only final and non-final states. In Step 2, we distinguish any two states of the same class for which there exists a letter leading them to states from different classes. Step 2 is repeated until no new class is constructed, see Fig. 2.6.

2.3 Chomsky Grammars

Formal languages do not stop at the boundary of Reg; the concept of *grammars* for generating languages allows to classify a larger variety of language families in which Reg will be embedded.

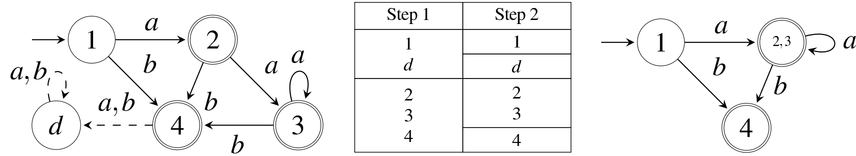

Fig. 2.6 Minimization: A modified DFA of Fig. 2.4 with the dead state d to be minimized, computation of the states, and the minimal automaton without the dead state d.

2.3.1 Type 0 Grammars and Languages

We begin with the most general type of grammars.

Definition 2.6. A *type 0 grammar* is a tuple $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$, where Σ is the *terminal* alphabet, $\mathscr{V}$ is the *non-terminal* alphabet (set of *variables*), $\mathsf{S} \in \mathscr{V}$ is the *axiom* (initial variable), $\mathsf{P} \subseteq (\Sigma \cup \mathscr{V})^* \mathscr{V} (\Sigma \cup \mathscr{V})^* \times (\Sigma \cup \mathscr{V})^*$ is a finite set of *rules* (productions) where at least one nonterminal appears on the left-hand side. A sentential form $\beta \in (\Sigma \cup \mathscr{V})^*$ is *derived* from a sentential form $\alpha \in (\Sigma \cup \mathscr{V})^*$, written $\alpha \Rightarrow \beta$, if there exists $(\alpha_2, \beta_2) \in \mathsf{P}$ such that $\alpha = \alpha_1 \alpha_2 \alpha_3$ and $\beta = \alpha_1 \beta_2 \alpha_3$.

Rules $(\alpha, \beta) \in P$ are also written as $\alpha \rightarrow \beta$ (read α *is rewritten by* β).

Definition 2.7. For a type 0 grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ and a sentential form $\alpha \in (\Sigma \cup \mathscr{V})^*$, let $\stackrel{*}{\Rightarrow}$ denote the reflexive and transitive closure of $\Rightarrow$. The *language generated by* α is the set $L_G(\alpha) = \{u \in \Sigma^* \mid \alpha \stackrel{*}{\Rightarrow} u\}$. The language generated by G, denoted $L(G)$, is defined as the set $L_G(\mathsf{S})$. A language is type 0 if it is generated by a type 0 grammar.

Example 2.4. The following type 0 grammar $G = (\{a\}, \{S, D, X, F, Y, Z\}, S, P)$ with P defined as follows generates the language $\{a^{2^n} \mid n > 0\}$.

$$\begin{array}{llll} \mathsf{S} \rightarrow DXaF & Xa \rightarrow aaX & XF \rightarrow YF & XF \rightarrow Z \\ aY \rightarrow Ya & DY \rightarrow DX & aZ \rightarrow Za & DZ \rightarrow \varepsilon \end{array}$$

Note that the alphabet and the set of variables can be extracted from the productions, so the set of productions characterizes the whole grammar. One possible derivation of this grammar is $\mathsf{S} \Rightarrow DXaF \Rightarrow DaaXF \Rightarrow DaaZ \Rightarrow DaZa \Rightarrow DZaa \Rightarrow aa$.

2.3.2 Type 1: Context-sensitive

We move ”up” in the hierarchy by restricting the productions to be monotonic in the length of the sub-words, that is, the generated word can never be shortened.

Definition 2.8. A grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ is *context-sensitive* (or *type 1*) if for all $(\alpha, \beta) \in \mathsf{P}$, $|\alpha| \leqslant |\beta|$. A language is context-sensitive (type 1 or in Cse) if it is generated by a type 1 grammar.

Example 2.5. The following is a type 1 grammar that generates the language $\{xx \mid x \in \{a,b\}^*\}$:

$$\begin{array}{llll} \mathsf{S} \to aAS \mid bBS \mid T & Aa \to aA & Ba \to aB & Ab \to bA \\ Bb \to bB & BT \to Tb & AT \to Ta & T \to \varepsilon \end{array}$$

A derivation chain for *abab* is

$$\mathsf{S} \Rightarrow aAS \Rightarrow aAbBS \Rightarrow aAbBT \Rightarrow abABT \Rightarrow abATb \Rightarrow abTab \Rightarrow abab\,.$$

Definition 2.9. A context-sensitive grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ is in *normal form (NF)* if every rule is of the form $\alpha_1 X \alpha_2 \to \alpha_1 \beta \alpha_2$ with $X \in \mathscr{V}$ and $\beta \neq \varepsilon$.

Theorem 2.7. *Every type 1 language can be generated by a context-sensitive grammar in NF.*

Example 2.6. A type 1 NF grammar for $\{a^{2^n} \mid n > 0\}$:

$$\begin{array}{lllll} \mathsf{S} \to aTa & T \to XA & XY \to Xa & Xa \to AAa & ZY \to ZX \\ \mathsf{S} \to aa & T \to AA & XY \to ZY & ZA \to AAA & aA \to aa \end{array}$$

2.3.3 Type 2: Context-Free Languages and Pushdown Automata

Note that a context-sensitive grammar in NF rewrites a variable X by a nonempty word β according to contexts α_1 and α_2. The next level is reached by loosing the information about these contexts.

Definition 2.10. A grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ is *context-free* or type 2 if $\mathsf{P} \subseteq \mathscr{V} \times (\Sigma \cup \mathscr{V})^*$. The class of languages generated by context-free grammars is denoted Cfr. A language is *linear* if it is generated by a context-free grammar with $\mathsf{P} \subseteq \mathscr{V} \times (\Sigma^* \cup \Sigma^* \mathscr{V} \Sigma^*)$.

Lemma 2.2 (Fundamental Lemma). *If $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ is context-free, then for all $\alpha_1, \alpha_2, \beta \in (\Sigma \cup \mathscr{V})^*$ and $n \geqslant 0$*

$$\alpha_1 \alpha_2 \stackrel{n}{\Rightarrow} \beta \iff \begin{cases} \alpha_1 \stackrel{n_1}{\Rightarrow} \beta_1 \\ \alpha_2 \stackrel{n_2}{\Rightarrow} \beta_2 \end{cases},$$

with $\beta = \beta_1 \beta_2$ and $n = n_1 + n_2$.

Example 2.7. The languages $L = \{a^n b^n \mid n \geqslant 0\}$ and the *nth Dyck language* of well-formed expressions with n brackets are in Cfr.

Definition 2.11. For a context-free grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$, a *derivation tree* is a tree labeled by $\mathscr{V} \cup \Sigma$ such that every interior node is $\mathscr{V}$-labeled, and if the sons of a node labeled x are labeled $\alpha_1, \ldots, \alpha_k$, then $(x, \alpha_1 \ldots \alpha_k)$ is a rule in P.

Example 2.8. Consider the language $\{a^n b^n \mid n \geqslant 0\}$ generated by a context-free grammar with rules $\mathsf{S} \to A\mathsf{S}b$, $A \to a$, and $\mathsf{S} \to \varepsilon$. A derivation tree of the derivation $\mathsf{S} \Rightarrow A\mathsf{S}b \Rightarrow a\mathsf{S}b \Rightarrow ab$ is shown in Fig. 2.7.

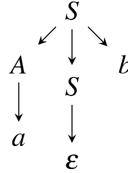

Fig. 2.7 Derivation tree of the word *ab*.

Lemma 2.3. *If* $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ *is a context-free grammar, then* (i) *for all* $x \overset{*}{\Rightarrow} \alpha$ *there exists a derivation tree with root x and yield* α*;* (ii) *if t is a derivation tree of G, then there exists a derivation* $root(t) \overset{*}{\Rightarrow} yield(t)$*.*

The class Cfr of context-free languages satisfies the following property of the same type as Lemma 2.1 for regular languages.

Lemma 2.4 (Pumping Lemma). *For every* $L \in \mathsf{Cfr}$ *there exists* $N \geqslant 0$ *such that for all* $w \in L$ *with* $|w| \geqslant N$, $w = \alpha u \beta v \gamma$*, with* $|uv| > 0$ *and* $|u\beta v| \leqslant N$*, and* $\forall n \in \mathbb{N}:$ $\alpha u^n \beta v^n \gamma \in L$*.*

As in the regular case, the pumping lemma allows to show *non*-inclusion of a language in Cfr; for instance, one can immediately see that $L_1 = \{a^n b^n c^n \mid n \geqslant 0\}$ is *not* context-free.

Closure Properties of Cfr**:** From the fact that L_1 is not context-free, we obtain that neither Cfr nor linear languages are closed under $\cap$; in fact, consider languages $\{a^n b^n c^p \mid n, p \geqslant 0\}$ and $\{a^p b^n c^n \mid n, p \geqslant 0\}$ whose intersection is the language L_1. On the other hand, Cfr *is* closed under $\cdot$ and $()^*$, as we will see below.

Pushdown Automata

As for regular languages, there exists a system model to recognize context-free languages.

Definition 2.12. A *pushdown automaton* is a tuple $A = (Q, \Sigma, Z, T, q_0, Q_m)$, where Q is a finite state set, Σ is an input alphabet, Z is a stack alphabet, $T \subseteq ZQ \times (\Sigma \cup \{\varepsilon\}) \times Z^* Q$ is a finite set of transitions, $q_0 \in ZQ$ is the initial configuration, and $Q_m \subseteq Q$ is the set of final states. The pushdown automaton A is called *real-time* if it is ε-free. The configurations of A are the words $hqw \in Z^* Q \Sigma^*$, where h is the content of the stack, q is the current state, and w is the unread part of the input.

Example 2.9. Consider the configuration $\beta pw = A\gamma pax$ from the first part of Fig. 2.8. After the execution of transition $(Ap, a, \alpha q)$, the new configuration is $\alpha\gamma qx$. We denote this transition by $A\gamma pax \xrightarrow{a} \alpha\gamma qx$.

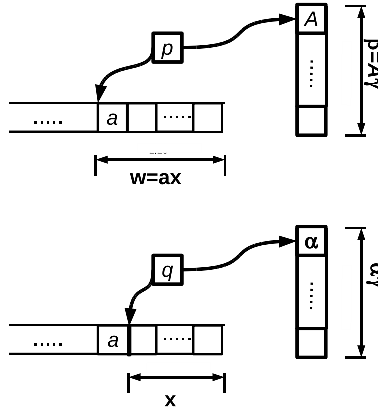

Fig. 2.8 Pushdown automaton executing $(Ap, a, \alpha q)$.

We have several notions of K-acceptance, with different values for K. Let $L(A) = \{w \in \Sigma^* \mid \exists q_0 \xrightarrow{w} h,\ h \in K\}$, where

- $K = Z^* Q_m$: *final state* acceptance.
- $K = Q$: *empty stack* acceptance.
- $K = Q_m$: *empty stack and final state* acceptance.

Proposition 2.1. *Let $L \subseteq \Sigma^*$.*

1. *If L is K-accepted by a PDA A, there exists another PDA A' that accepts L by final state.*
2. *For every PDA A, the empty stack language L_A can be generated by a context-free grammar G that can be effectively constructed from A.*
3. *Conversely, for any context-free grammar G, a PDA A can be constructed that accepts $L(G)$ with empty stack.*

The closure for $\cdot$ and $()^*$ can be shown by combining the accepting automata.

2.3.4 Type 3 Languages

Finally, on the last level, we find regular languages as a subclass of Cfr by further restrictions of context-free rules so they cannot generate more than one nonterminal at a step which is always the last (resp. the first) symbol of any sentential form.

Definition 2.13. A grammar $G = (\Sigma, \mathscr{V}, \mathsf{S}, \mathsf{P})$ is *left-linear* if $\mathsf{P} \subseteq \mathscr{V} \times (\Sigma^* \cup \mathscr{V}\Sigma^*)$, and *right-linear* if $\mathsf{P} \subseteq \mathscr{V} \times (\Sigma^* \cup \Sigma^*\mathscr{V})$.

Proposition 2.2. *A language is regular if and only if it is generated by a left-linear (or right-linear) grammar.*

Example 2.10. There are linear languages that are not regular; in fact, languages $\{a^n b^n \mid n \geqslant 0\}$ and $\{a^n b^n c^p \mid n, p \geqslant 0\}$ are linear.

2.3.5 The Chomsky Hierarchy

The following chain of inclusions summarizes the hierarchical relations between the language classes introduced here:

$$\mathsf{Reg} \subsetneq \mathsf{Cfr} \subsetneq \mathsf{Cse} \subsetneq \mathsf{REN}\,. \tag{2.1}$$

Here, we use REN to denote the class of type 0 languages, for reasons given below.

2.4 Turing Machines and Decidability

We now leave the realm of grammar-generated languages and turn to the most fundamental model of computation, that of *Turing machines or TMs*. A TM consists of $k \geqslant 1$ two-way infinite tapes, each of whose cells are $\mathbb{Z}$-indexed. Every cell contains a symbol which is either blank (\$) or a letter from an auxiliary alphabet Σ. The k read/write heads (one for each tape) controlled by its internal state and the tape content move along the tapes. Formally, we have:

Definition 2.14. A *k-tape Turing machine* (TM) is a tuple $\mathscr{M} = (Q, \Sigma, \$, T, q_0, Q_m)$, where *(i)* Q is a finite state set, *(ii)* Σ is an alphabet of *tape symbols*, *(iii)* $\$ \in \Sigma$ is the *blank* symbol; set $\tilde{\Sigma} \triangleq \Sigma \setminus \{\$\}$; *(iv)* $q_0 \in Q$ is the initial state, *(v)* $Q_m \subseteq Q$ is the set of final states, and *(vi)* $T \subseteq Q \times \Sigma^k \times \tilde{\Sigma}^k \times \{L, R, S\}^k \times Q$ is a set of *instructions* (q, s, s', m, q') for reading from/writing to the tapes, moving the k heads (left, right, stay) and state change. The TM $\mathscr{M}$ is *deterministic* (DTM) if $(q_1, s_1, s'_1, m_1, q'_1), (q_2, s_2, s'_2, m_2, q'_2) \in T$ with $(s'_1, m_1, q'_1) \neq (s'_2, m_2, q'_2)$ implies $(q_1, s_1) \neq (q_2, s_2)$.

There exist several structurally different definitions for TMs in the literature. In particular, there may be $k > 1$ tapes, one-sided tapes (i.e., the indices are restricted to $\mathbb{N}$), etc. All these models can be shown to be *equivalent* in the sense that the computability/decidability notions they define coincide. Of course, the concrete constructions proving each of these equivalences involve non-trivial modifications of alphabets, state spaces, etc., and produce very different machines.

The dynamics of TMs are illustrated in Fig. 2.9.

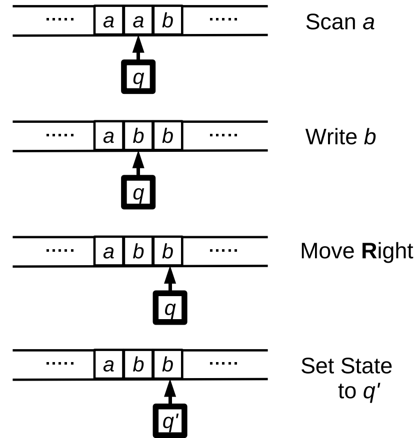

Fig. 2.9 One-Tape Turing Machine executing (q,a,b,R,q').

Definition 2.15 (TM Languages). Let $\mathscr{M} = (Q,\Sigma,\$,T,q_0,Q_m)$ be a Turing machine. A global state/configuration (q,w,z) of $\mathscr{M}$ consists of the current state q, the content w of the tape cells that are *not* \$, and the index $z \in \mathbb{Z}^k$ of the current positions of heads. The *input* is the non-\$ tape content in the initial configuration. Computation $\mathscr{M}(x)$ of $\mathscr{M}$ on input x *halts* if it is finite and ends in a final state. Input x is *accepted* if $\mathscr{M}(x)$ halts in an *accepting* state, and *rejected* if $\mathscr{M}(x)$ halts in a *non-accepting (rejecting)* state. The set $T(\mathscr{M}) \subseteq \tilde{\Sigma}^*$ of inputs accepted by $\mathscr{M}$ is the *language accepted by* $\mathscr{M}$. Languages accepted by TMs are called *recursively enumerable*; denote the class of recursively enumerable languages by REN. A language L such that there exists a TM $\mathscr{M}_L$ which always halts when given a finite sequence of symbols from the alphabet of the language as input, and such that $T(\mathscr{M}_L) = L$, is called *recursive* or *decidable*; the class of these languages is denoted REC.

The following is a small selection of results from a vast collection.

1. $L \subseteq \Sigma^*$ is type 0 if and only if it is recursively enumerable (and therefore REN appears rightfully in (2.1) above).
2. $\mathsf{Cse} \subsetneq \mathsf{REC} \subsetneq \mathsf{REN} \neq 2^{\Sigma^*}$.
3. For any multi-tape machine $\mathscr{M}$ there exists a single-tape machine $\mathscr{M}'$ that simulates $\mathscr{M}$, i.e., $L(\mathscr{M}) = L(\mathscr{M}')$, using only quadratically more computation time.
4. Many more equivalences hold; e.g., every DTM can be simulated by a DTM having only two states.

The following definition establishes TMs as a model for computation.

Definition 2.16 (Computability). For an input x such that $\mathscr{M}(x)$ halts, denote by $f_{\mathscr{M}}(x)$ the tape content on halting. Any function $f : \tilde{\Sigma}^* \to \tilde{\Sigma}^*$ for which there exists a TM $\mathscr{M}$ with $f(w) = f_{\mathscr{M}}(w)$, for all $w \in \tilde{\Sigma}^*$, is called *computable*.

Theorem 2.8 (Decidability). *A language $L \subseteq \Sigma^*$ is* decidable *if and only if its characteristic function $\chi_L : \Sigma^* \to \{0,1\}$ is computable.*

We can now state the celebrated *Church-Turing-Rosser Thesis.*

EVERYTHING COMPUTABLE CAN BE COMPUTED BY A TM.

This thesis is not a claim about a mathematical object, and hence cannot have a proof. Instead, it postulates that the computability notion defined via the TM model is the *"right"* model. The reader willing to follow the thesis is therefore invited to continue with the remainder of the chapter.

2.4.1 Universal Turing Machine

One can see the TM-based computational model as a "programmable machine", in the sense that one strives to find a *universal* TM $\mathscr{U}$ whose inputs consist of a description $\langle \mathscr{M}, x\rangle$ of a TM $\mathscr{M}$ and an input x such that $\mathscr{M}(x)$ is simulated on $\mathscr{U}$.

Such a construction is actually possible: $\tilde{\Sigma}_{\mathscr{U}}$ encodes the i-th symbol of $\tilde{\Sigma}_{\mathscr{M}}$ by the binary description of i; similarly for $Q_{\mathscr{M}}$ and x; L, R, S are $00, 01, 10$; parts of the encoding are separated by special symbols, e.g. ";". Three tapes are needed: one for the description of $\mathscr{M}$ and x, one for simulating $\mathscr{M}'$s tape, and one for the description of $\mathscr{M}'s$ current state and the symbol read by $\mathscr{M}'$s tape head. If $\mathscr{M}$ halts, then $\mathscr{U}$ halts with the final state of $\mathscr{M}$. If the input is *not* a valid description of a TM and an input, $\mathscr{U}$ never halts.

Again, we give a selection of results on the *halting problem*.

Theorem 2.9. *The universal language $L_{\mathscr{U}} = \{\langle \mathscr{M}, w\rangle \mid w \in L(\mathscr{M})\}$ is recursively enumerable.*

Proof. Proof by construction of a universal TM. □

Let $\langle \mathscr{M}\rangle$ denote a string encoding of $\mathscr{M}$. The following theorem says that there exists a TM which can construct a description of $\mathscr{M}$ from its encoding.

Theorem 2.10. *The function $\langle \mathscr{M}\rangle \mapsto \langle \mathscr{M}, \langle \mathscr{M}\rangle\rangle$ is computable.*

Theorem 2.11. *If L and L^c are recursively enumerable, then L and L^c are recursive.*

Proof. A new TM to decide L runs TMs for both L and L^c in parallel. As each word w is either in L or in L^c, the machine always halts. □

Theorem 2.12. *$L^c_{\mathscr{U}}$ is not recursively enumerable, so $L_{\mathscr{U}}$ is not recursive.*

Proof. The proof is by contradiction. Let $\mathscr{M}_N$ be a TM with $L(\mathscr{M}_N) = L^c_{\mathscr{U}}$. Construct a TM D such that $L(D) = \{\langle \mathscr{M}\rangle \mid \langle \mathscr{M}\rangle \notin L(\mathscr{M})\}$. Apply $\mathscr{M}_N$ to $\langle D, \langle D\rangle\rangle$. Then we have the absurdity $\langle D\rangle \in L(D) \Longleftrightarrow \langle D\rangle \notin L(D)$. □

Theorem 2.13. *The* halting problem *is undecidable, that is, the language $L_{halt} = \{\langle \mathscr{M}, x\rangle \mid \mathscr{M}(x)$ halts$\}$ is recursively enumerable but not recursive.*

Proof. The proof is by contradiction as in the above. □

2.4.2 Decidable and Undecidable Problems

Reduction is a strategy for showing that a problem $P \triangleq (D \in S)$ is undecidable. It consists of two steps:

1. Take an undecidable problem $P' \triangleq (D' \in S')$;
2. Reduce P' to P by giving a recursive function $f : S' \to S$ such that for all $D' \in S'$, $(D' \in S') \iff (f(D) \in S)$.

Examples of Decidable Problems

- The word problem for a DFA A: $w \in L(A)$?
- Emptiness problem for a DFA A: $L(A) = \emptyset$?
- Equivalence of regular expressions.
- Emptiness problem, word problem, finiteness problems for context-free languages.
- The word problem for context-sensitive languages.

Examples of Undecidable Problems

Rice's theorem: For $Q \subseteq \mathsf{REN}$ such that $\emptyset \neq Q \neq \mathsf{REN}$, the problem $P \triangleq (L \in Q)$ is undecidable.
Post's Correspondence Problem (PCP): Given two sequences of words $(u_1, \ldots, u_n)$, $(v_1, \ldots, v_n) \in (\Sigma^*)^n$. The question is whether there exist $k > 0$ and indices $i_1, \ldots, i_k$ such that $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$.

The PCP is a very popular undecidable problem in the literature for its reducibility to other problems since it often allows for shorter or more elegant proofs than reducing directly the halting problem. There exist several proofs of undecidability of the PCP; a reduction of the halting problem goes as follows (see, e.g., [15]): Given a TM, construct pairs (u_i, v_i) as blocks, each of which has two fields, with name u_i in the top field and v_i in the bottom field so that aligning these blocks, the top and bottom rows form words that code the moves of the TM, which halts exactly if and only if the PCP can be solved.

Turning to other formal languages, the *emptiness problem* for type 1 grammars is undecidable. Furthermore, for G_1, G_2 context-free and $R \in \mathsf{Reg}$, the following problems are known to be undecidable:

- $L(G_1) = \Sigma^*$?
- $L(G_1) \cap L(G_2) = \emptyset$?
- $L(G_1) \subseteq L(G_2)$?
- $L(G_1) = L(G_2)$?
- $R \subseteq L(G_1)$?
- $L(G_1) \in \mathsf{Reg}$?
- Is $L(G_1) \cap L(G_2)$ context-free?
- Is $L(G_1)^c$ context-free?

2.5 Complexity Classes

While the previous section asked whether a given problem is possible to decide, we now ask, for problems known to be decidable, how difficult it is to solve. That is:

- What type of computation is needed (deterministic/nondeterministic TM)?
- What is the time and space required?

Let us formalize this.

Definition 2.17. A non-decreasing function $f : \mathbb{N} \to \mathbb{N}$ is *constructible* if there exists a TM $\mathscr{M}$ such that for input x with $|x| = n$, $\mathscr{M}(x)$ produces output of length $f(x)$

- in time (number of transitions) $\mathscr{O}(n + f(n))$
- and space $\mathscr{O}(f(n))$ (number of cells visited).[1]

For fixed f, language L is in

- $\mathsf{NTIME}(f)$ ($\mathsf{TIME}(f)$) if there exists $n_0 \in \mathbb{N}$ and a TM (DTM) $\mathscr{M}$ that takes time $f(n)$ for every $x \in L$ such that $|x| \geqslant n_0$.
- $\mathsf{NSPACE}(f)$ ($\mathsf{SPACE}(f)$) if there exists $n_0 \in \mathbb{N}$ and a TM (DTM) $\mathscr{M}$ that takes space $f(n)$ for every $x \in L$ such that $|x| \geqslant n_0$.

We have some obvious inclusions:

$$\begin{aligned}
\mathsf{TIME}(f) &\subseteq \mathsf{NTIME}(f) \\
\mathsf{SPACE}(f) &\subseteq \mathsf{NSPACE}(f) \\
\forall\, k > 0 :\ \mathsf{TIME}(kf) &\subseteq \mathsf{TIME}(f) \\
\mathsf{NTIME}(kf) &\subseteq \mathsf{NTIME}(f) \\
\forall\, k > 0 :\ \mathsf{SPACE}(kf) &\subseteq \mathsf{SPACE}(f) \\
\mathsf{NSPACE}(kf) &\subseteq \mathsf{NSPACE}(f)
\end{aligned}$$

Other inclusions require more work, see [8, 15]. For every constructible f,

$$\begin{aligned}
\mathsf{NTIME}(f) &\subseteq \mathsf{SPACE}(f) \\
\mathsf{NSPACE}(f) &\subseteq \bigcup_{i \in \mathbb{N}} \mathsf{TIME}\left(i^{f+\log(n)}\right)
\end{aligned}$$

The following list collects the classes most currently used, obtained by taking canonical functions for f, i.e. $f(n) \equiv \log_2(n)$ or $f(n) \equiv n^k$.

$$\begin{aligned}
\mathsf{LOGSPACE} &\triangleq \mathsf{SPACE}(\log_2(n)) \\
\mathsf{NLOGSPACE} &\triangleq \mathsf{NSPACE}(\log_2(n)) \\
\mathsf{P} = \mathsf{PTIME} &\triangleq \bigcup_{k \in \mathbb{N}} \mathsf{TIME}(n^k)
\end{aligned}$$

[1] $\mathscr{O}(f(n))$ denotes the class of functions that do not grow faster than f, i.e., $\mathscr{O}(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{N}, \forall n \in \mathbb{N} : |g(n)| \leqslant c \cdot |f(n)|\}$; see the literature for more details.

$$\begin{aligned}
\mathsf{NP} = \mathsf{NPTIME} &\triangleq \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k)\\
\mathsf{PSPACE} &\triangleq \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(n^k)\\
\mathsf{NPSPACE} &\triangleq \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)\\
\mathsf{EXP} = \mathsf{EXPTIME} &\triangleq \bigcup_{k\in\mathbb{N}} \mathsf{TIME}(2^{n^k})\\
\mathsf{NEXP} = \mathsf{NEXPTIME} &\triangleq \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{n^k}).
\end{aligned}$$

We have the hierarchy of

$$\mathsf{LOGSPACE} \subseteq \mathsf{NLOGSPACE} \subseteq \mathsf{PTIME} \subseteq \mathsf{NPTIME} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE}.$$

Here, some inclusions may not be proper ones, see below.

2.5.1 Reduction

Reduction is used to prove that a particular problem belongs to a complexity class.

Definition 2.18. A *reduction* from $f : A \to \{\mathbf{tt},\mathbf{ff}\}$ to $f' : B \to \{\mathbf{tt},\mathbf{ff}\}$ is a computable function $h : A \to B$ such that $\forall a \in A$, $f'(h(a)) = f(a)$.

Complete and Hard Problems

Definition 2.19. A problem P in a class C is *complete* for C if every P' in C reduces to P via a function h computable in C.

One frequently encounters the notion of NP-hard problems. A problem P is NP-hard if there exists an NP-hard problem P' that is polynomial-time Turing-reducible to P. Note that a problem can be NP-hard without being NP-complete; this is the case of the (undecidable!) *halting problem* discussed above. Thus, an NP-hard problem is NP-complete if it in addition belongs to NP.

Finally, the problem “Is $\mathsf{P} = \mathsf{NP}$?” is probably the most famous open question in computer science.

Examples from NP

For more details, the reader is referred to [3].

Satisfiability (SAT)

Let ϕ be a propositional formula, i.e., connecting atoms $p_i \in A_t$ by $\neg, \wedge, \vee$. The question is: Is ϕ *satisfiable*, i.e., is there $\nu : A_t \to \{\mathbf{tt}, \mathbf{ff}\}$ such that $\nu(\phi) = \mathbf{tt}$?

- SAT is in NP: a TM "guesses" ν and computes $\nu(\phi)$ in polynomial time.
- SAT is NP-complete; to see this, let $L \in \mathsf{NP}$ and let $\mathscr{M}$ be a TM that solves L in (nondeterministic) polynomial time. One constructs a formula that holds if and only if $\mathscr{M}$ halts and accepts (Cook 1971).

Hamiltonian Circuit (HC)

Take a directed graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$. Does there exist a permutation σ of $\{1, \ldots, n\}$ such that $\forall\ 1 \leqslant i \leqslant n-1$, $(v_{\sigma(i)}, v_{\sigma(i+1)}) \in E$?

- HC is in NP since a TM can solve HC by trial-and-error in polynomial time.
- HC is NP-complete, which can be shown by reducing SAT to it.

The Knapsack Problem

Given $v_1, \ldots, v_n, w \in \mathbb{N}$. Is there $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} v_i = w$?
NP-completeness of Knapsack is shown by reduction of (a variant of) SAT to it.

Weighted Path (WP)

In a weighted graph $G = (V, E, p)$ with $p : E \to \mathbb{N}$ and $u, v \in V$, find a path from u to v such that the edge weights sum to a fixed value N.
NP-completeness follows here because Knapsack can be reduced to WP.

The Class PSPACE

First of all, we have the following result:

Theorem 2.14 (Savitch 1970). *The problem of accessibility in a directed graph with n vertices can be solved in space* $\mathscr{O}((\log n)^2)$.

As a corollary, we have that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. The following problems are from PSPACE:

- Satisfiability of a Boolean formula with quantifiers ($\exists$, $\forall$).
- Universality of regular languages: $L = \Sigma^*$?

Note that $L = \emptyset$ is in PTIME but $L = \Sigma^*$ is not!

2.6 Further Reading

An introduction to automata and formal language theory can be found in [5, 10, 15, 16], and advanced material in [11, 14, 17]. More details on complexity, computability, and decidability can be found in [1, 2, 7, 8, 9], and a long list of NP-complete problems can be found in [3].

Acknowledgements

This work was supported by the European Commission's 7th Framework Program grant No. INFSO-ICT-224498, by the GAČR grant No. P202/11/P028, and by RVO: 67985840.

References

1. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer Verlag, Berlin, 2nd edition, 2001.
2. D. P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, New York, 1994.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, 1990.
4. S. Ginsburg. *Algebraic and Automata-theoretic Properties of Formal Languages*. North-Holland, Amsterdam, 1975.
5. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Boston, MA, 3rd edition, 2006.
6. G. Jirásková and T. Masopust. Complexity in union-free regular languages. In *Proc. of DLT 2010*, LNCS 6224, pages 255–266, Berlin, 2010. Springer Verlag.
7. N. D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press Cambridge/Mass and London/UK, 1997.
8. D. C. Kozen. *Automata and Computability*. Springer Verlag, Berlin, 1997.
9. C. Papadimitriou. *Computational Complexity*. Addison Wesley, Boston, MA, 1993.
10. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1-3. Springer Verlag, Berlin, 1997.
11. J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, New York, 2009.
12. A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.
13. A. Salomaa. *Computation and Automata*. Cambridge University Press, Cambridge, 1985.
14. J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, New York, 2008.
15. M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, 2nd edition, 2005.
16. T. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley, Boston, MA, 3rd edition, 2005.
17. A. Szepietowski. *Turing Machines with Sublogarithmic Space*. Springer Verlag, Berlin, 1994.

Index

A

Alphabet 2
Automaton
  ε-automaton 4
  canonical 8
  deterministic 3
  minimal 8
  nondeterministic 4
  pushdown 11
    real-time 11

C

Church-Turing-Rosser Thesis 15
Concatenation 2

D

Decidability 15
Derivation tree 10

E

Event set 2

F

Factor 2
Function
  computable 14
  constructible 17

G

Grammar 8
  context-free 10
  context-sensitive 9
  left-linear 13
  linear 10
  right-linear 13
  type 0 9
  type 1 9
  type 2 10

H

Homomorphism 2

I

Infix 2

L

Language 2
  accepted 3
  decidable 14, 15
  recognizable 4
  recursive 14
  recursively enumerable 14
  regular 3
Letters 2

M

Mirror image 2
Morphism 2

P

Powers 2
Prefix 2
Problem

complete 18
halting problem 15
Post’s correspondence 16
Projection 2
Pumping lemma 7, 11

Q

Quotient 2

R

Reduction 18
Regular expressions 3
Reversal 2

S

String 2
Sub-word 2
Substitution 2
Suffix 2
Symbols 2

T

Turing machine 13
deterministic 13
universal 15

W

Word 2
empty 2
length 2