Automata and Formal Languages

Fall 2007

Juhani Karhumäki

Preface

Theory of formal languages (or automata) constitutes a cornerstone of theoretical computer science. However, its origin and motivation come from different sources:

1. *Switching circuits* as models for electrical engineers.

2. *Grammars* as models for the structure of natural languages (Chomsky, 1956).

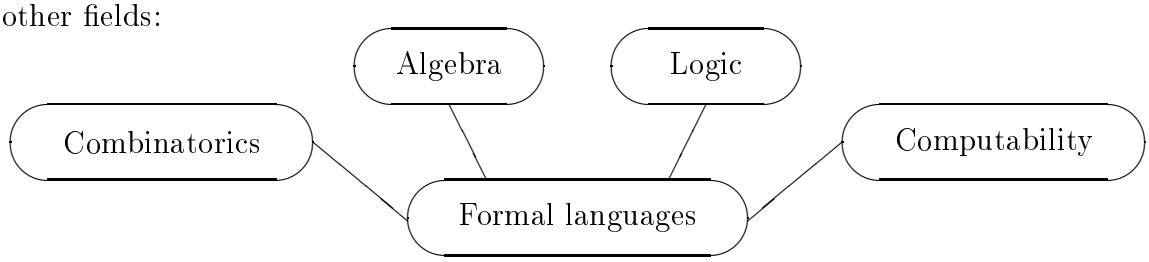
3. Models for biological phenomena:
   Neural networks which lead to finite automata (McCulloh, Pitts, 1943).
   Lindenmayer systems as models for the growth of organisms (Lindenmayer, 1968).

4. Models in different parts of theory of programming languages:
   parsing, compiling, text editing, ...

5. Models for mathematical (and philosophical) questions of computability (Turing, 1936; Post).

The above list also provides examples of applications of formal languages. Other newer application areas are *cryptography* and *computer graphics*.

Formal language theory is part of discrete mathematics having connections to many other fields:



In this course we concentrate on languages (e.g. sets of words) described by *finite automata*, *context-free grammars* and *Turing machines*.

Literature:

Berstel, J.: *Transductions and Context–Free Languages*, Teubner, 1979.

Davis, M. and Weyuker, E.J.: *Computability, Complexity and Languages*, Academic Press, 1983.

Harrison, M.A.: *Introduction to Formal Language Theory*, Addison–Wesley, 1978.

Hopcroft, J.E. and Ullman, J.O.: *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, 1979.

Kozen, D.: *Automata and Computability*, Springer–Verlag, 1997.

Lewis, H.R. and Papadimitriou, C.H.: *Elements of Theory of Computation*, Prentice Hall, 1981.

Salomaa, A.: *Formal Languages*, Academic Press, 1973.

Sipser, M.: *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

Wood, D.: *Theory of Computation*, John Wiley, 1994.

Contents

Chapter 1

Preliminaries

1.1 Basic notions of words and languages

First we fix some notions and notations of words:

Alphabet Σ: nonempty (finite) set of symbols, like $\Sigma = \{a, b\}$.

Word w: a sequence of symbols, like $(a, b, a) = aba$.

Σ^* (resp. Σ^+): the set of all finite (resp. finite nonempty) words.

Language L: a set of words, $L \subseteq \Sigma^*$.

Empty word 1: the sequence of 0 symbols.

Catenation or *product* of words:

$$a_1 \cdots a_n \cdot b_1 \cdots b_m = a_1 \cdots a_n b_1 \cdots b_m.$$

Clearly, this is an associative operation, so that $(\Sigma^*, \cdot)$ (resp. $(\Sigma^+, \cdot)$) is a monoid (resp. semigroup) having 1 as the unit element. Moreover, they are *free*, i.e. each word has the unique presentation as the product of letters. They are called the *free monoid* and *free semigroup* generated by Σ.

Let $w, u \in \Sigma^*$, $\Delta \subseteq \Sigma$, $a \in \Sigma$ and $L, K \subseteq \Sigma^*$. We set:

Length of $w = |w|$: total number of letters in w; $|1| = 0$.

$|w|_a$: number of a's in w.

Alphabet of w: $\mathrm{alph}(w) = \{a \in \Sigma \mid |w|_a \geq 1\}$.

Factors : u is a *factor* of w (resp. *left factor* or *prefix*, *right factor* or *suffix*) if there exists words x and y such that

$$w = xuy \text{ (resp. } w = uy \text{ , } w = xu).$$

Factors are *proper* if they are different from w. In the case of left or right factors we write:

$$u = wy^{-1} \quad \text{and} \quad u = x^{-1}w,$$

as well as

$$u \leq w \quad (\text{resp. } u < w)$$

if u is a (resp. proper) left factor of w.

Reverse of $w = a_1 \cdots a_n,\ a_i \in \Sigma$: $\quad w^R = a_n \ldots a_1$.

Factorization of w: any sequence $u_1, \ldots, u_n$ of words such that

$$w = u_1 \cdots u_n. \tag{1.1}$$

(1.1) is *L-factorization* iff each $u_i \in L$. It is natural to write

$$L^* = \{u_1 \cdots u_n \,|\, n \geq 0,\ u_i \in L\},$$

and

$$L^+ = \{u_1 \cdots u_n \,|\, n \geq 1,\ u_i \in L\}.$$

Hence, each word in L^* has *at least* one L-factorization. If there exists only one such factorization then L is a *code* and it *freely generates* the monoid L^* (Indeed, L^* is a monoid, and a submonoid of Σ^*).

n:th power of w: $\quad w^0 = 1,\ w^n = (w^{n-1})w = w(w^{n-1})$.

Finally, $w \in \Sigma^+$ is *primitive* iff it is not a proper power of any word:

$$w = u^n \implies n = 1 \text{ (and hence } u = w).$$

Most of the above notions extend in a natural way to languages:

$$\mathrm{alph}(L) = \bigcup_{w \in L} \mathrm{alph}(w),$$

$$L^{-1}K = \{u^{-1}w \,|\, u \in L,\ w \in K\}, \text{ etc.}$$

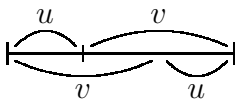
Next we prove a few basic combinatorial properties of words:

Theorem 1.1. *Words $u, v \in \Sigma^*$ commute, i.e. $uv = vu$, iff they are powers of a some word, i.e. there exists z such that $u, v \in z^*$.*

Proof. $\Leftarrow$: Clear.
$\Rightarrow$: By induction on $|uv|$.
Case $|uv| = 0$ is clear: $u = v = 1$. Assume that $|uv| = k$. Now

$$uv = vu \tag{1.2}$$

which can be illustrated as:



If $|u| = |v|$, then necessarily $u = v$, and we can choose $z = u$. So by symmetry, we may assume $|u| < |v|$. Hence there exists $t \in \Sigma^+$ such that

$$v = ut.$$

If $u = 1$ we can choose $z = v$. Otherwise we can write (1.2) in the form

$$uut = utu,$$

or equivalently,

$$ut = tu. \tag{1.3}$$

Since $|u| \neq 0$, $|ut| < |uv|$ and we can apply induction hypothesis to (1.3) to conclude that there exists z such that $u, t \in z^*$. Hence $u, v \in z^*$, too. □

Theorem 1.1 characterizes commutation of two words. Similarly we can characterize the property that words u and v are *conjugates*, i.e. for some words p and q, $u = pq$ and $v = qp$.

Theorem 1.2. *Words $u, v \in \Sigma^+$ are conjugates iff*

$$\exists z \ : \quad uz = zv \tag{1.4}$$

iff

$$\exists z, p, q \ : \quad u = pq \ , \ v = qp \ \ \textit{and} \ \ z \in p(qp)^*. \tag{1.5}$$

Proof. Clearly, if u and v are conjugates, they satisfy (1.5), and conversely (1.5) implies that u and v are conjugates. So it remains to be proved that (1.4) and (1.5) are equivalent.
(1.5) $\Rightarrow$ (1.4): Clear. Indeed, if $u = pq$, $v = qp$ and $z = p(qp)^n$, $n \geq 0$, then

$$uz = pq\,p(qp)^n = p\,qp\,(qp)^n = p(qp)^n\,qp = zv.$$

(1.4) $\Rightarrow$ (1.5): Assume that $uz = zv$. Then for all $n \geq 1$:

$$u^n z = u^{n-1}uz = u^{n-1}zv \underset{\text{hyp.}}{\overset{\text{ind.}}{=}} zv^{n-1}v = zv^n.$$

Now, choose n such that

$$n|u| \geq |z| > (n-1)|u|$$

and consider the equation

$$u^n z = zv^n. \tag{1.6}$$

Then necessarily

$$z = u^{n-1}p \quad \text{and} \quad zq = u^n \quad \text{for some } p \text{ and } q.$$

Now

$$u^n = zq = u^{n-1}pq \ , \ \ \text{so that } u = pq$$

and, by (1.6) and above,

$$v^n = qz = q(pq)^{n-1}p = (qp)^n \ , \ \ \text{so that } v = qp.$$

This completes the proof. □

The proof of Theorem 1.1 translates straightforwardly to

Theorem 1.3. *Any words $u, v \in \Sigma^*$ satisfying a nontrivial identity are powers of a some word.*

Theorem 1.3 is a version of so–called Defect Theorem (cf. Lothaire: *Combinatorics of words*). It also has an interesting corollary

Theorem 1.4. *For each word $u \in \Sigma^+$ there exists a unique primitive word ρ such that $u = \rho^n$ for some $n \geq 1$.*

Proof. Clearly, there exists at least one such ρ. Assume that both ρ and ρ_1 are such words, say $\rho^n = u = \rho_1^m$. Then ρ and ρ_1 satisfy a nontrivial identity

$$\rho^{2n} = xx = \rho_1^{2m}$$

so that they are powers of a word, and hence by the primitiveness $\rho = \rho_1$. □

The word $\rho = \rho(u)$ of the previous theorem is called the *primitive root* of u.

As the final example of combinatorial properties of words we mention (without a proof) the following basic periodicity lemma. Here u^ω denotes the infinite word $uu\ldots$.

Theorem 1.5. *Let $u, v \in \Sigma^+$. If u^ω and v^ω have a common prefix of length $|u| + |v| - \gcd(|u|, |v|)$, then $\rho(u) = \rho(v)$.*

A weaker form of Theorem 1.5 is as follows:

Corollary 1.6. *If $x \leq u^\omega$, $x \leq v^\omega$ and $|x| \geq |u| + |v|$, then $\rho(u) = \rho(v)$.*

Now, we turn from words to languages, i.e. sets of words. There are a number of natural operations on languages. For languages $L, K \subseteq \Sigma^*$ we define:

union:	$L \cup K$	Boolean operations
intersection:	$L \cap K$	
complement:	$\Sigma^* \setminus L = L'$	
difference:	$L \setminus K = \{w \mid w \in L, w \notin K\}$	
catenation or product:	$LK = \{uv \mid u \in L, v \in K\}$	
left quotient:	$L^{-1}K = \{u^{-1}v \mid u \in L, v \in K\}$	
power:	$L^0 = \{1\},\ L^n = (L^{n-1})L$	
iteration or Kleene star:	$L^* = \bigcup_{i \geq 0} L^i = \{u_1 \cdots u_n \mid n \geq 0, u_i \in L\}$	
1-free iteration or Kleene plus:	$L^+ = \bigcup_{i \geq 1} L^i = \{u_1 \cdots u_n \mid n \geq 1, u_i \in L\}$.	

Operations union, catenation and iteration are called *rational*. Indeed, they correspond arithmetic operations sum, product and inverse ($L^* = \{1\} \cup L \cup L^2 \ldots \leftrightarrow (1 - L)^{-1}$).

Two other important operations are *morphic* and *inverse morphic image* of a language:

$$h(L) = \{h(u) \mid u \in L\} \subseteq \Delta^*$$

and

$$h^{-1}(L) = \{v \mid h(v) \in L\} \subseteq \Sigma^*,$$

where $h : \Sigma^* \to \Delta^*$ is a *morphism*, i.e. mapping satisfying

$$h(uv) = h(u)h(v)\ ,\ \forall u, v \in \Sigma^*.$$

We can easily conclude a number of identities on languages, for example

$$\begin{aligned} (LM)N &= L(MN), \\ L(M \cup N) &= LM \cup LN, \\ (L^*)^* &= L^*L^* = L^*, \text{ etc.} \end{aligned}$$

1.2 Specifications of languages and language families

An obvious problem is: *How to specify a language*? In the case of finite languages the simplest way is to *make a list* of all words of a language. For an infinite language this would not lead to a finite description.

Is is also worth noting that although Σ is finite, Σ^* is denumerable and, hence the number of subsets of Σ^*, i.e. languges over Σ is nondenumerable. Consequently, we can "effectively" describe only very few of all possible languages.

The three methods used in this course to describe languages are as follows:

I. Via certain *operations*,

II. Via *acceptance* by a device,

III. Via *generation* by a grammar.

I. We say that a family $\mathcal{L}$ of languages (over Σ) is *closed* under operation φ, if whenever φ is applied to a language in $\mathcal{L}$ the result is also in $\mathcal{L}$. Now, one way to define a family $\mathcal{L}$ is to fix certain family of *initial languages* and certain *closure operations*, and say that $\mathcal{L}$ constitutes of those languages which are obtained by applying a finite number of times these operations to initial languages. Or more concretely:

Definition 1.1. The family of *rational* languages over Σ, in symbols $\mathrm{Rat}(\Sigma)$, is defined:

(i) $\emptyset \in \mathrm{Rat}(\Sigma)$ and $\{a\} \in \mathrm{Rat}(\Sigma)$, for $a \in \Sigma$,

(ii) if $L_1, L_2 \in \mathrm{Rat}(\Sigma)$, then $L_1 \cup L_2 \in \mathrm{Rat}(\Sigma)$,

(iii) if $L_1, L_2 \in \mathrm{Rat}(\Sigma)$, then $L_1 L_2 \in \mathrm{Rat}(\Sigma)$,

(iv) if $L \in \mathrm{Rat}(\Sigma)$, then $L^* \in \mathrm{Rat}(\Sigma)$,

(v) $\mathrm{Rat}(\Sigma)$ is the smallest family satisfying (i)-(iv).

Clearly, (v) can be replaced by

(vi) $\mathrm{Rat}(\Sigma)$ contains only those languages which are obtained from languages in (i) by applying operations (ii)-(iv) a finite number of times.

Thus each rational language is obtained by applying operations union, catenation and iteration finitely many times to singleton languages and the empty language. Consequently, we can associate with a rational language a sequence of applications of (ii)-(iv), in other words, an expression, called *rational expression*, which describes the initial languages and applications of the operations. In order to avoid unnecessary parenthesis in these operations we agree the *preference* of the operations to be

$$\text{iteration, catenation, union.}$$

Moreover we identify $\{a\}$ and a. Then, for example

$$ab^* \cup ba = \big(a(b^*)\big) \cup (ba)\ ,\ \emptyset^* = \{1\},$$

and the identity

$$a(aa)^* \cup (aa)^* = a^*\ (= \{a\}^*)$$

shows that the representation is not unique.

Definition 1.2. Formally *rational expressions* are defined as:

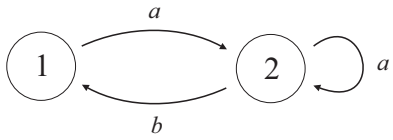
(i) $\emptyset$ and a, for $a \in \Sigma$, are rational expressions,

(ii-iv) if α and β are rational expressions, so are $(\alpha \cup \beta)$, $(\alpha\beta)$ and (α^*).

(v) these are all rational expressions.

Now languages defined by rational expressions are defined in a natural way: $\emptyset$ defines empty language, a defines $\{a\}$ and $(\alpha \cup \beta)$ defines the union of those defined by α and β, etc.

II. We take here only a very simple example. Consider the set of paths in the following labeled graph leading from the node 1 into itself. The sequence of labels encountered forms a word over $\{a, b\}$, and hence the above rule defines a language, which is $(a^+b)^*$.



III. In this method an initial symbol, as well as (substitution) rules how from a word new words are derived and the language generated consists of those words (of certain type) which are obtained from the initial symbol by these rules.

For example, if the initial symbol is S and the rules are "S can be replaced by aSb or 1", formally $S \to aSb$, $S \to 1$, then the set of words in $\{a, b\}^*$ obtained is $\{a^n b^n \mid n \geq 0\}$. Or more formally:

Definition 1.3. *Chomsky grammar*, or *grammar* in short, is a quadruple

$$\mathcal{G} = (V, \Sigma, \mathcal{P}, S),$$

where

- V is an alphabet containing *terminal* and *nonterminal* alphabets, Σ and $N = V \backslash \Sigma$,
- $\mathcal{P}$ is a finite set of *productions*, which are of the form

$$\alpha \longrightarrow \beta\ , \quad \alpha \in V^*NV^* \text{ and } \beta \in V^*,$$

- S is the *initial symbol*.

Define a relation $\Rightarrow_{\mathcal{G}}$ in $V^* \times V^*$ as follows:

$$u \Rightarrow_{\mathcal{G}} v \quad \text{iff} \quad \exists u', u'', \alpha, \beta \in V^* : u = u'\alpha u'',\ v = u'\beta u'' \text{ and } \alpha \to \beta \in \mathcal{P}.$$

Let $\Rightarrow^*_{\mathcal{G}}$ be the *transitive and reflexive closure* of $\Rightarrow_{\mathcal{G}}$, i.e.

$$u \Rightarrow^*_{\mathcal{G}} v \quad \text{iff} \quad u = v \text{ or } \exists k \geq 1 \text{ and } u_1, \ldots, u_k \in V^* : \underbrace{u = u_1 \Rightarrow_{\mathcal{G}} u_2 \Rightarrow_{\mathcal{G}} \ldots \Rightarrow_{\mathcal{G}} u_k = v}_{= D}.$$

Now, the *language generated by* $\mathcal{G}$ is

$$L(\mathcal{G}) = \{w \in \Sigma \mid S \Rightarrow^*_{\mathcal{G}} w\}.$$

For brevity, we often write $\Rightarrow$ instead of $\Rightarrow_{\mathcal{G}}$. Further if $u \Rightarrow_{\mathcal{G}} v$ (resp. $u \Rightarrow^*_{\mathcal{G}} v$) we say that u *derives directly* (resp. *derives*) v according to $\mathcal{G}$. A *derivation* of v from u is the above sequence D.

Example 1.1. Consider grammar $(\{a, b, S, S'\}, \{a, b\}, \mathcal{P}, S)$, where $\mathcal{P} = \{S \to S', S \to aSb, S' \to S'b, S' \to 1\}$. Then we have

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaS'bb \Rightarrow aaS'bbb \Rightarrow aabbb \in L(\mathcal{G}),$$

and

$$S \Rightarrow S' \Rightarrow S'b \Rightarrow S'bb \Rightarrow bb \in L(\mathcal{G}).$$

It is not difficult to see that

$$L(\mathcal{G}) = \{a^n b^m \mid m \geq n \geq 0\}.$$

Example 1.2. Consider grammar $\mathcal{G} = (\{a, \#, t, t', S\}, \{a\}, \mathcal{P}, S)$, where $\mathcal{P}$ consists of

$$\begin{array}{rcl} S & \longrightarrow & \#ta\#, \\ ta & \longrightarrow & aat, \\ t\# & \longrightarrow & t'\# \mid t', \\ at' & \longrightarrow & t'a, \\ \#t' & \longrightarrow & \#t \mid 1. \end{array}$$

With two exceptions next step of any derivation is unique. Based on this one can conclude that $L(\mathcal{G}) = \{a^{2^n} \mid n \geq 1\}$.

Our goal is to define different families of languages by grammars. This is achieved by restricting the form of productions leading to so–called *Chomsky hierarchy*

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0 \tag{1.7}$$

defined below. As we shall see each of these families can be defined as families accepted by different types of automata, as well.

Family	Form of productions	Type of languages	Corresponding automata
$\mathcal{L}_0$	no restriction	RE, recursively enumerable	Turing machine
$\mathcal{L}_1$	$\alpha A\gamma \to \alpha\beta\gamma,\ S \to 1$ $A \in N,\ \alpha, \beta, \gamma \in V^*,\ \beta \neq 1$	CS, context–sensitive	linearly bounded automaton (lba)
$\mathcal{L}_2$	$A \to \beta,\ A \in N, \beta \in V^*$	CF, context–free	pushdown automaton (pda)
$\mathcal{L}_3$	$A \to \alpha \mid \alpha B$ (right linear) $A, B \in N,\ \alpha \in \Sigma^*$	Rat or Reg rational or regular	finite automaton (FA)

Table 1.1: Classification of grammars

Of course, languages in $\mathcal{L}_i$ in table 1.1 on page 7 can be called *type* i languages.

We shall see that the hierarchy (1.7) is strict. Actually, there are many possibilities to refine the Chomsky hierarchy. The diagram 1.2 on page 8 illustrates these possibilities (becomes clearer later).

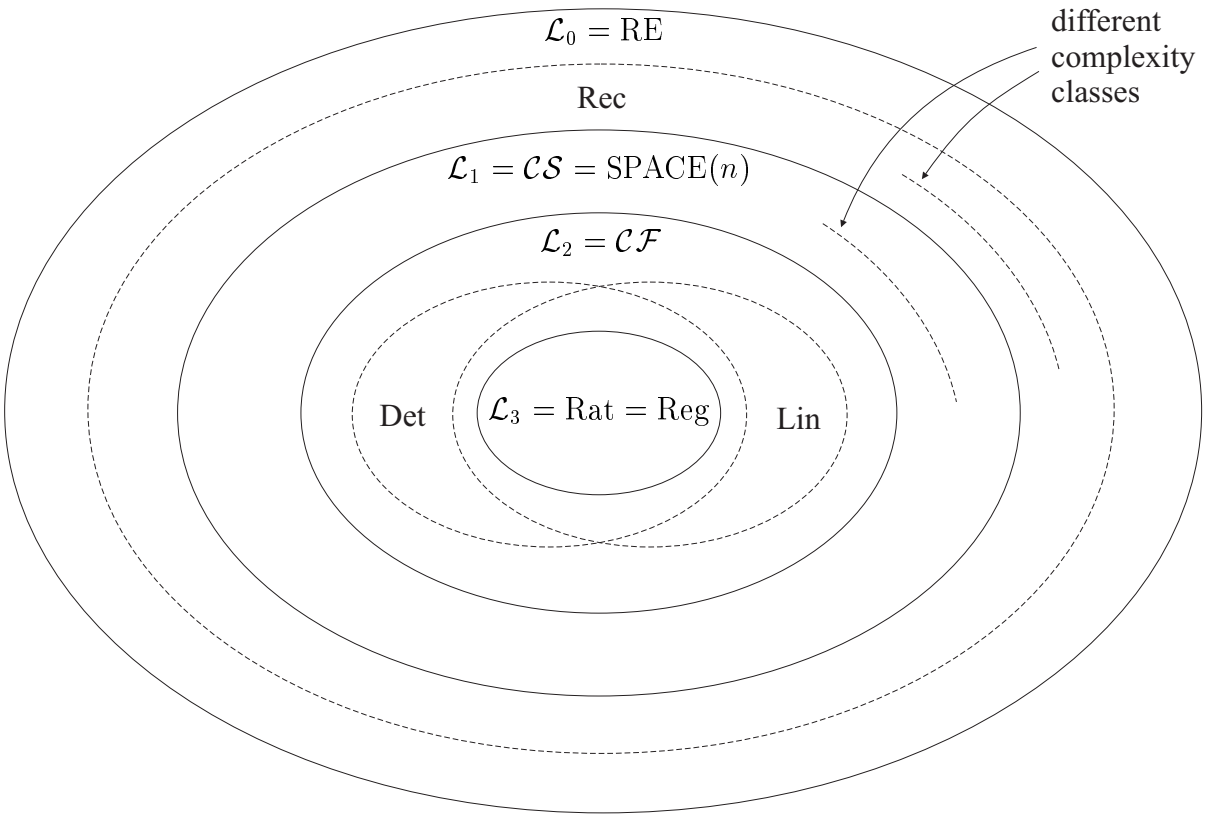


Figure 1.1: Refined Chomsky hierarchy

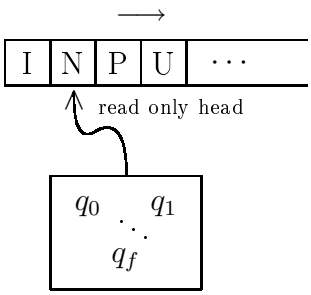
Chapter 2

Regular languages

2.1 Finite automata

Family of regular languages is very basic in formal language theory. Mathematically regular languages are natural extensions of finite languages, and from the computer science point of view, they correspond languages which can be recognized by *finite memory* devices.

Informally, finite automaton can be described as follows: It consists of an input tape, a reading head and a finite number of internal states. It reads the input symbol by symbol, and in a step the automaton can change its internal state based on the symbol and the current state. Initially the automaton is in a specified initial state, and it accepts the input, if after reading it the automaton is in some of specified final states.



Definition 2.1. Formally, *deterministic finite automaton*, DFA for short, is a quintuple

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F),$$

where

(i) Q is a finite set of *states*

(ii) Σ is a finite *input alphabet*,

(iii) $\delta : Q \times \Sigma \to Q$ is a partial *transition function*,

(iv) $q_0 \in Q$ is the *initial* state,

(v) $F \subseteq Q$ is a set of *final* states.

The (partial) mapping $\delta : Q \times \Sigma \to Q$ is extended to a (partial) mapping $Q \times \Sigma^* \to Q$ (which is still denoted by δ) as follows:

$$\begin{aligned} \delta(q, 1) &= q \\ \delta(q, wa) &= \delta\big(\delta(q, w), a\big) \quad \forall w \in \Sigma^*,\ a \in \Sigma. \end{aligned}$$

DFA $\mathcal{A}$ *accepts* or *recognizes* the language

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

Further a language $L \subseteq \Sigma^*$ is *regular* or *recognizable* iff it is accepted by a DFA. The family of all regular languages over Σ is denoted by $\mathrm{Reg}(\Sigma)$ or $\mathrm{Rec}(\Sigma)$.

Now, the relation $w = a_1 \cdots a_n \in L(\mathcal{A})$, with $a_i \in \Sigma$, means that there exist states $q_1, \ldots, q_n$ such that

$$\delta(q_{i-1}, a_i) = q_i \quad \text{for } i = 1, \ldots, n \ \text{ and } \ q_n \in F.$$

This can be illustrated as follows:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n \in F \tag{2.1}$$

or

$$(q_0, a_1 \cdots a_n) \vdash (q_1, a_2 \cdots a_n) \vdash \cdots \vdash (q_{n-1}, a_n) \vdash (q_n, 1). \tag{2.2}$$

More briefly, the above illustrations, which show how $\mathcal{A}$ *accepts* w or how $\mathcal{A}$ *computes on* w, can be written:

$$q_0 \xrightarrow{a_1 \cdots a_n} q_n \in F$$

or

$$(q_0, a_1 \cdots a_n) \vdash^* (q_n, 1).$$

Note that $\vdash^*$ here denotes the reflexive transitive closure of $\vdash$ which corresponds to one step derivation in $\mathcal{A}$ (cf. grammars on page 6).

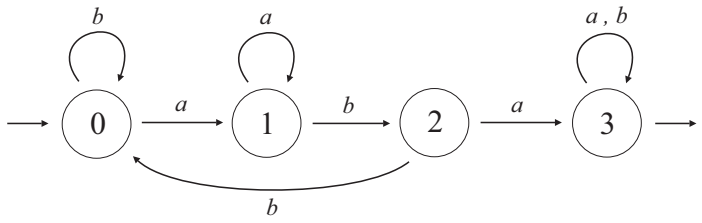
As a conclusion, $\mathcal{A}$ accepts exactly those words which lead from the initial state to a final state. Corresponding paths are called *successful*.

Example 2.1. Let $\mathcal{A} = (\{0,1,2,3\}, \{a,b\}, \delta, 0, \{3\})$, where δ is given as:

δ:

Q \ Σ	a	b
$\to 0$	1	0
1	1	2
2	3	0
$\leftarrow 3$	3	3

$\mathcal{A}$ can be illustrated as:



As illustrated in this example, DFA can be given as a *transition table* or as a labelled *transition graph*, where edge $p \xrightarrow{a} q$ corresponds transition $\delta(p,a) = q$, and initial and final states are given by incoming and outgoing arrows, respectively.

It is not difficult to conclude that

$$L(\mathcal{A}) = \{a,b\}^* aba \{a,b\}^* = \big\{ w \in \{a,b\}^* \mid w \text{ contains } aba \text{ as a factor} \big\}.$$

This becomes even more illustrative if the states 0, 1, 2 and 3 are replaced by 1, a, ab and aba — then the state remembers how much of aba is already found!

Remark 2.1. DFA $\mathcal{A}$ is *complete* iff δ is a total function. Then also its extension to $Q \times \Sigma^*$ would be total. A given $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *can be completed* as follows:
Let $\mathcal{A}^c = (Q \cup \{g\}, \Sigma, \delta', q_0, F)$, where

$$\delta'(q,a) = \begin{cases} g & \text{if } \delta(q,a) \text{ is not defined,} \\ \delta(q,a) & \text{otherwise,} \end{cases} \qquad \text{and} \qquad \delta'(g,a) = g \quad \forall a \in \Sigma.$$

Obviously, $\mathcal{A}^c$ is complete, and *equivalent* with $\mathcal{A}$, i.e. $L(\mathcal{A}) = L(\mathcal{A}^c)$. Indeed, each computation of $\mathcal{A}$ can be carried out in $\mathcal{A}^c$, and if a computation in $\mathcal{A}$ stops, due to the lack of the next transition, $\mathcal{A}^c$ moves to g, so–called *garbage state*, from where no final state is reachable. Due to this remark, there is normally no need to pay attention to the completeness of DFA.

In DFA each word, either accepted or not, has a unique (if at all) computation. This is not true in nondeterministic automata.

Definition 2.2. A *nondeterministic finite automaton*, NFA for short, is like a DFA except that δ and q_0 are replaced by E and Q_0:

(iii′) $E \subseteq Q \times \Sigma \times Q$ is a *transition relation*,

(iv′) $Q_0 \subseteq Q$ is a set of *initial states*.

A language *accepted* by an NFA $\mathcal{A}$ is

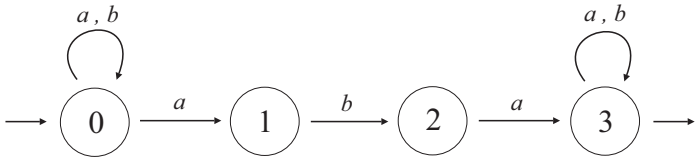
$$L(\mathcal{A}) = \{w \mid \exists a_1, \ldots, a_n \in \Sigma, q_0, \ldots, q_n \in Q : \ w = a_1 \cdots a_n, \\ q_0 \in Q_0, \ q_n \in F \text{ and } (q_{i-1}, a_i, q_i) \in E \text{ for } i = 1, \ldots, n\}.$$

Obviously, notations (2.1) and (2.2), as well as the transition table and the transition graph representations suit for nondeterministic finite automata, too. For example, $p \xrightarrow{w} q$ means that w causes in $\mathcal{A}$ a transition from p to q. In these terms

$$L(\mathcal{A}) = \{w \mid \exists p \in Q_0, q \in F : \ p \xrightarrow{w} q \text{ in } \mathcal{A}\}.$$

As a conclusion, a word is accepted by NFA $\mathcal{A}$ iff there exists *at least* one accepting path from an initial state into a final one.

Example 2.1 (continued). The language of Example 2.1 is accepted by an NFA:



Remark 2.2. The extension (iv') is not essential. For an NFA $\mathcal{A} = (Q, \Sigma, E, Q_0, F)$ we define $\mathcal{A}' = (Q \cup \{q_0\}, \Sigma, E \cup E', \{q_0\}, F)$, where q_0 is a new state and $E' = \{(q_0, a, q) \mid \exists p \in Q_0 : (p, a, q) \in E\}$. Obviously, $\mathcal{A}'$ contains only one initial state, and $L(\mathcal{A}) = L(\mathcal{A}')$.

Theorem 2.1. *A language L is regular iff it is accepted by an NFA.*

Proof. It has to be shown that each language L accepted by an NFA, say $\mathcal{A} = (Q, \Sigma, E, Q_0, F)$, is accepted by a DFA as well.

We construct a DFA $\mathcal{B} = (P, \Sigma, \delta, q_0, G)$ as follows:

$$\begin{aligned} P &= 2^Q = \text{the power set of } Q, \\ \delta(H, a) &= \{q \in Q \mid \exists p \in H : \ (p, a, q) \in E\} \quad \text{for } H \in P, \ a \in \Sigma, \\ q_0 &= Q_0, \\ G &= \{H \in P \mid H \cap F \neq \emptyset\}. \end{aligned}$$

Clearly, $\mathcal{B}$ is deterministic (and complete) and we show:

Claim. $\delta(H, w) = \{q \in Q \mid \exists p \in H : \ p \xrightarrow{w} q \text{ in } \mathcal{A}\}$ *for all* $w \in \Sigma^*$, $H \in P$.

This is shown by induction on $|w|$.

$|w| = 0$: Now

$$\delta(H, 1) \overset{\text{def. of } \delta}{=} H \overset{\text{by convention!}}{=} \{q \in Q \mid \exists p \in H : \ p \xrightarrow{1} q \text{ in } \mathcal{A}\}.$$

Induction step:

$$\begin{aligned}
\delta(H,ua) &= \delta\big(\delta(H,u),a\big) \overset{\text{i.h.}}{=} \delta\big(\overbrace{\{q \in Q \mid \exists p \in H :\ p \xrightarrow{u} q \text{ in } \mathcal{A}\}}^{=Q'},a\big)\\
&= \{r \in Q \mid \exists q \in Q' :\ q \xrightarrow{a} r \text{ in } \mathcal{A}\}\\
&= \{r \in Q \mid \exists q \in Q', p \in H :\ p \xrightarrow{u} q \text{ and } q \xrightarrow{a} r \text{ in } \mathcal{A}\}\\
&= \{r \in Q \mid \exists p \in H :\ p \xrightarrow{ua} r \text{ in } \mathcal{A}\}.
\end{aligned}$$

Now the theorem follows easily:

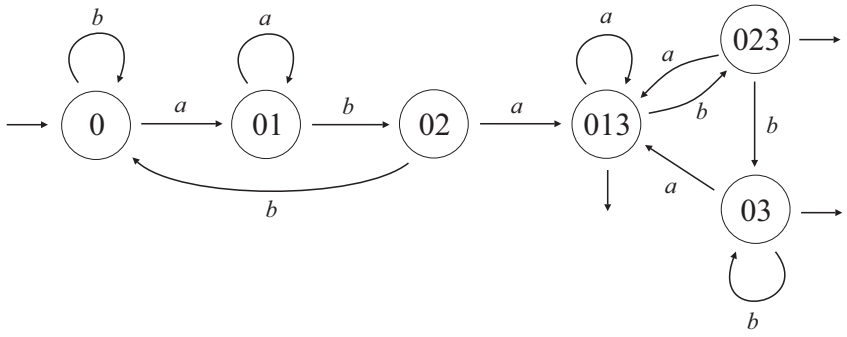
$$\begin{aligned}
L(\mathcal{B}) &= \{w \in \Sigma^* \mid \delta(Q_0,w) \in G\} = \{w \in \Sigma^* \mid \delta(Q_0,w) \cap F \neq \emptyset\}\\
&= \{w \in \Sigma^* \mid \exists p \in Q_0, q \in F :\ p \xrightarrow{w} q \text{ in } \mathcal{A}\} = L(\mathcal{A}).
\end{aligned}$$

□

Remark 2.3. The construction of the proof of Theorem 2.1 is called *subset construction*. Clearly, it increases the number of states exponentially. It can be shown that this exponential blow up cannot be avoided in general, cf. Figure 2.1 on the next page.

Remark 2.4. For certain purposes NFA´s are much more suitable than DFA´s. For example the fact that the reverse of a regular language L, that is $L^R = \{w^R \mid w \in L\}$, is also regular. Indeed, in NFA accepting L it is enough to chance the directions of the arrows and initial and final state sets in order to obtain an NFA for L^R.

Example 2.1 (continued). A subset construction for an automaton of page 11 yields:

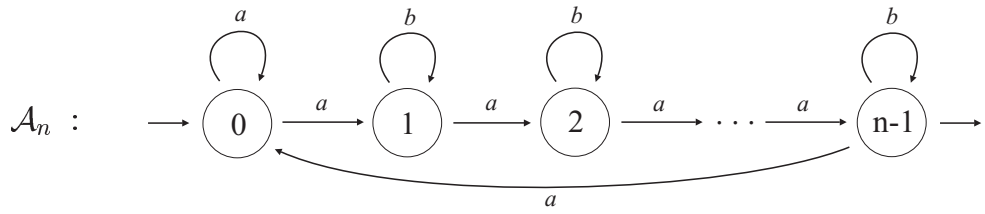


This should be compared to the DFA of page 10. Note that if the construction is done step–by–step the transitions need to be defined only for those states which are *reachable* from the initial state.

We can always assume that an FA contains cycles from a state into itself labeled by the empty word (cf. proof of Theorem 2.1). Indeed, such cycles have no affect to the language generated. On the other hand transitions $p \xrightarrow{1} q$ cannot be added without affecting the language generated. In a generalized automaton also such transitions are allowed.

Definition 2.3. A *generalized finite automaton* $\mathcal{A} = (Q, \Sigma, E, I, F)$, GFA for short, is like an NFA except that transition relation is now of the form:

(iii″) $E \subseteq Q \times \Sigma^* \times Q$ is a *finite* transition relation.



n=4 : Subset construction

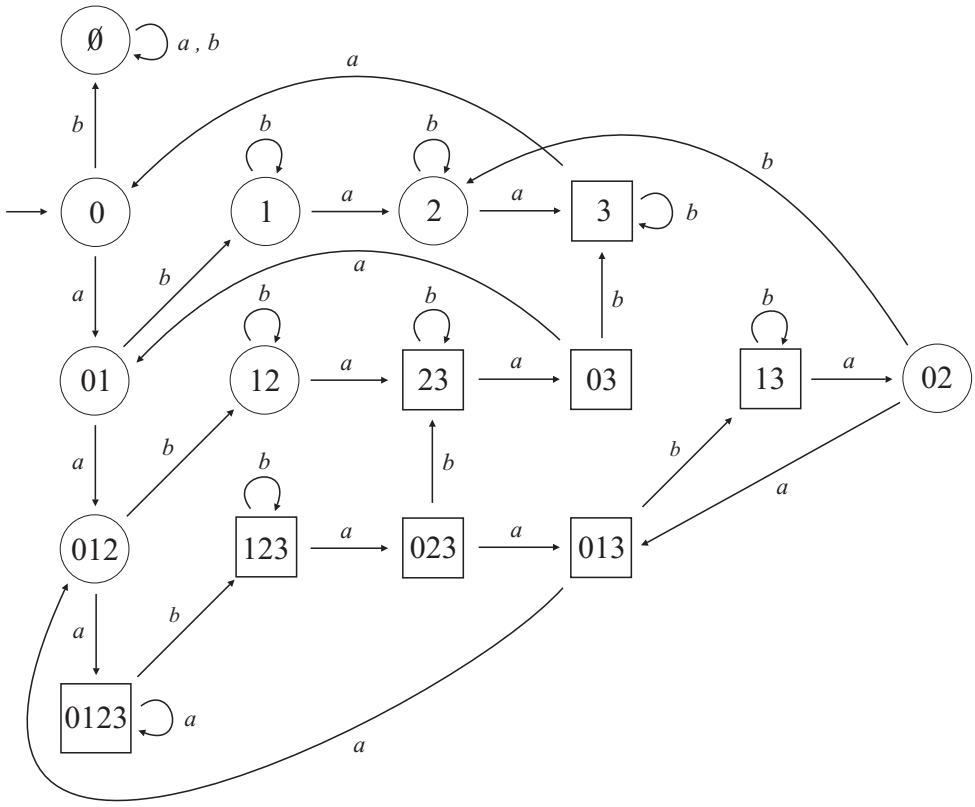
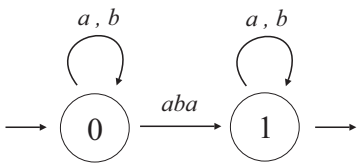


Figure 2.1: The smallest DFA for $L(\mathcal{A}_4)$ is of the size $2^{|Q_4|}$.

Remark 2.5. As for NFA for GFA one can always find an equivalent automaton of the same type having only one initial state.

Example 2.1 (continued). Clearly the language of this example is accepted by 2-state GFA:



Next we show that even this generalization of an FA does not increase the accepting power.

Theorem 2.2. *A language L is regular iff it is accepted by a GFA.*

Proof. We have to show, by Theorem 2.1, that the language accepted by a GFA is accepted by an NFA as well. By the above remark we can assume that the GFA contains only one initial state. So let $L = L(\mathcal{A})$ for such a GFA $\mathcal{A}$.

$\mathcal{A}$ may contain two types of "illegal" transitions:

$$p \xrightarrow{w} q\ , \quad \text{with } w = a_1 \cdots a_n,\ n \geq 2,\ a_i \in \Sigma,$$

or

$$p \xrightarrow{1} q\ , \quad \text{with } p \neq q. \qquad (*)$$

Clearly, a transition of the former type can be eliminated by replacing it with the transitions

$$p \xrightarrow{a_1} p_1\ ,\ p_1 \xrightarrow{a_2} p_2\ , \ldots,\ p_{n-1} \xrightarrow{a_n} q,$$

where $p_1, \ldots, p_{n-1}$ are new states not in $\mathcal{A}$. Obviously, the language accepted is not changed, so that after a finite number of applications of this procedure we obtain a GFA accepting the same language and having no productions of this form. Consequently, we may assume that $E \subseteq Q \times (\Sigma \cup \{1\}) \times Q$.

In order to eliminate productions of the form $(*)$ we need an auxiliary notion: For a state p, its *1-closure*, $\mathrm{clos}(p)$ for short, is defined as follows:

$$\mathrm{clos}(p) = \{q \mid p \xrightarrow{1} q \text{ in } \mathcal{A}\}.$$

1-closure of p can be computed by the following procedure. Set

$$\begin{aligned} C_0(p) &= \{p\}, \\ C_{i+1}(p) &= C_i(p) \cup \{q \mid \exists r \in C_i(p) :\ (r, 1, q) \in E\}. \end{aligned}$$

Then clearly,

$$\mathrm{clos}(p) = \bigcup_{i \geq 0} C_i(p)$$

and

$$C_i(p) \subseteq C_{i+1}(p)\ , \quad \forall i \geq 0.$$

Therefore $\mathrm{clos}(p) = \bigcup_{i=0}^{i_0} C_i(p)$, where i_0 is the smallest index s.t. $C_{i_0-1}(p) = C_{i_0}(p)$.

Now, let a GFA $\mathcal{A}$ be $(Q, \Sigma, E, \{q_0\}, F)$, with $E \subseteq Q \times (\Sigma \cup \{1\}) \times Q$. We construct an NFA $\mathcal{A}' = (Q, \Sigma, E', Q_0, F)$ as follows:

$$\begin{aligned} Q_0 &= \mathrm{clos}(q_0), \\ (p, a, q) \in E',\ a \in \Sigma &\Leftrightarrow \exists r \in Q :\ (p, a, r) \in E,\ a \in \Sigma \text{ and } q \in \mathrm{clos}(r). \end{aligned}$$

We claim that

$$L(\mathcal{A}') = L(\mathcal{A}).$$

Inclusion $L(\mathcal{A}') \subseteq L(\mathcal{A})$ is clear: Each successful path in $\mathcal{A}'$

- starts from a state in Q_0, and
- leads through transitions in E' to a state in F.

The same word is accepted by $\mathcal{A}$ since

– by the definition of Q_0, $\mathcal{A}$ can go from q_0 to any state of Q_0 by reading the empty word,

– each transition of E' can be simulated in $\mathcal{A}$ by a sequence of transitions.

Conversely, the inclusion $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ follows since each computation of $\mathcal{A}$ can be factorized uniquely to the form

$$q_0 \xrightarrow{1} q_1 \xrightarrow{a_1} q_1' \xrightarrow{1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{1} q_n \xrightarrow{a_n} q_n' \xrightarrow{1} q, \quad \text{with } a_i \in \Sigma.$$

The dot line shows how this computation can be simulated in $\mathcal{A}'$. □

Remark 2.6. The proof of Theorem 2.2 is algorithmic, that is it provides an algorithm to construct for each GFA an equivalent NFA. The same applies to the proof of Theorem 2.1.

2.2 Properties of regular languages

In this section we establish several basic properties of regular languages, which are useful to show that certain languages are regular, or also to show that they are not. We start with *closure properties.*

Theorem 2.3. *The family of regular languages is closed under Boolean operations.*

Proof. Let $L_1, L_2 \subseteq \Sigma^*$ be accepted by DFA´s $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, say $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, q_0(\mathcal{A}_i), F_i)$, with $i = 1, 2$.

Union: $L_1 \cup L_2$ is accepted by a GFA $\mathcal{A}_\cup$ described as:

$\mathcal{A}_\cup :$ → q_0 —1→ $q_0(\mathcal{A}_1) \longrightarrow \cdots$ and q_0 —1→ $q_0(\mathcal{A}_2) \longrightarrow \cdots$ graphs of $\mathcal{A}_1$ and $\mathcal{A}_2$,

where q_0 is the initial state of $\mathcal{A}_\cup$, and $F_1 \cup F_2$ is its set of final states. Here we have to assume, as we can, that $Q_1 \cap Q_2 = \emptyset$. Now, by Theorem 2.2, $L_1 \cup L_2 \in \mathrm{Reg}$.

Complement: If $\mathcal{A}_1$ is complete, as we can assume, then $\Sigma^* \setminus L_1$ is accepted by the automaton $\mathcal{A}_1^C$ which is obtained from $\mathcal{A}_1$ by changing its final states to nonfinal, and conversely. Indeed, in a complete DFA each word has the unique computation which is successful in $\mathcal{A}_1$ iff it is not successful in $\mathcal{A}_1^C$.

Intersection: By de Morgan Law $L_1 \cap L_2 = \big(L_1^C \cup L_2^C\big)^C$ or directly:
Define $\mathcal{A}_\cap = (Q, \Sigma, \delta, q_0, F)$ as follows:

$$\begin{aligned} Q &= Q_1 \times Q_2, \\ \delta\big((q_1, q_2), a\big) &= \big(\delta_1(q_1, a), \delta_2(q_2, a)\big), \\ q_0 &= \big(q_0(\mathcal{A}_1), q_0(\mathcal{A}_2)\big), \\ F &= \{(q_1, q_2) \mid q_1 \in F_1,\ q_2 \in F_2\}. \end{aligned}$$

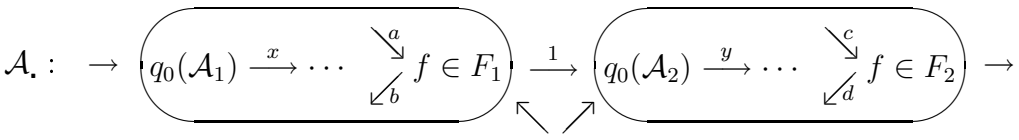
By construction a word w causes a successful computation in $\mathcal{A}_\cap$ iff it causes successful computations in $\mathcal{A}_1$ *and* $\mathcal{A}_2$. □

Theorem 2.4. *The family of regular languages is closed under rational operations.*

Proof. We use the notations of Theorem 2.3.

Union: Theorem 2.3.

Catenation: L_1L_2 is accepted by a GFA $\mathcal{A}_\bullet$ illustrated as follows:



Kleene star: If we add to $\mathcal{A}_1$ transitions $\{(p, 1, q_0) \mid p \in F\}$ we obtain an automaton accepting Kleene plus of L_1, i.e. L_1^+. So the result follows from the identity $L_1^* = L_1^+ \cup \{1\}$. □

Theorem 2.5. *The family of regular languages is closed under morphisms and inverse morphisms.*

Proof. Let $h : \Sigma^* \to \Delta^*$ be a morphism.

Morphism: If a regular language $L \subseteq \Sigma^*$ is accepted by a DFA $\mathcal{A}$, then a GFA $\mathcal{A}_h$ accepting $h(L) \subseteq \Delta^*$ is obtained from it by changing the input alphabet to Δ and transitions as follows:

$$p \xrightarrow{h(a)} q \quad \text{in } \mathcal{A}_h \qquad \text{iff} \qquad p \xrightarrow{a} q \quad \text{in } \mathcal{A}.$$

Inverse morphism: Let $L \subseteq \Delta^*$ be accepted by a FA $\mathcal{A}$. We modify $\mathcal{A}$ to a new automaton $\mathcal{A}_{h^{-1}}$ as follows:

$$p \xrightarrow{a} q \quad \text{in } \mathcal{A}_{h^{-1}} \qquad \text{iff} \qquad p \xrightarrow{h(a)} q \quad \text{is a path in } \mathcal{A}.$$

Then obviously $L(\mathcal{A}_{h^{-1}}) = h^{-1}(L)$. □

Remark 2.7. Note that the number of states of a DFA in the above construction for inverse morphisms does not grow while for morphisms it may grow.

Next we provide a tool to show that some languages are not regular.

Theorem 2.6 (Pumping Lemma). *Let L be accepted by an n-state FA $\mathcal{A}$. Then for each $w \in L$, with $|w| \geq n$, there exist words u, v and z such that*

$$w = uvz\ , \quad \textit{with } |uv| \leq n,\ v \neq 1$$

and moreover,

$$uv^*z \subseteq L.$$

Proof. Let $w = a_1 \cdots a_{|w|}$ with $a_i \in \Sigma$. Since $\mathcal{A}$ has only n states it follows from the pigeon hole principle that the sequence

$$q_0,\ q_1 = \delta(q_0, a), \ldots,\ q_n = \delta(q_0, a_1 \cdots a_n)$$

contains a repetition, i.e. there exist indices i and j, $0 \leq i < j \leq n$, such that $q_i = q_j$. Choose $u = a_1 \cdots a_i$, $v = a_{i+1} \cdots a_j$ and $z = a_{j+1} \cdots a_{|w|}$. Then the first claim follows.

Assuming that $w \in L$, we conclude from the fact $\delta(q_i, v) = q_j = q_i$, that $uv^*z \subseteq L$, as required. □

Remark 2.8. Sometimes the above Pumping Lemma is given in the following weaker forms:

(i) Each word w of length at least n accepted by an n-state FA $\mathcal{A}$ admits a factorization $w = uvz$, with $v \neq 1$, such that $uv^*z \subseteq L(\mathcal{A})$.

(ii) For each infinite regular language L, there exist words u, v and z, with $v \neq 1$, such that $uv^*z \subseteq L$.

Example 2.2. We claim that the languages

$$L_1 = \{a^n b^n \mid n \geq 1\}$$

and

$$L_2 = \left\{w \in \{a,b\}^* \mid |w|_a = |w|_b\right\}$$

are not regular. If L_1 would be regular, then by Theorem 2.6, we would find $m > 0$ and k such that

$$a^{k+mt}b^k \in L_1\ , \quad \text{for all } t \geq 0,$$

which is not the case.

To prove that $L_2 \notin \mathrm{Reg}$, we assume the contrary. Then, by Theorem 2.3,

$$L_2 \cap a^*b^* \in \mathrm{Reg}\,.$$

However, $L_2 \cap a^*b^* = L_1$.

Theorem 2.6 has also theoretical applications.

Corollary 2.7. *Let $\mathcal{A}$ be an n-state automaton. Then*

(i) $L(\mathcal{A}) \neq \emptyset$ *iff* $\exists w \in L(\mathcal{A}) :\ |w| < n.$

(ii) $L(\mathcal{A})$ *is infinite iff* $\exists w \in L(\mathcal{A}) :\ n \leq |w| < 2n.$

Proof. (i) $\Leftarrow$: Clear.
$\Rightarrow$: Let w be (some) shortest word in $L(\mathcal{A})$. If $|w| < n$ we are done; otherwise by the above Remark 2.8 (i) we can write $w = uvz$, $v \neq 1$ and $uz \in L(\mathcal{A})$, a contradiction to the minimality of w.

(ii) $\Leftarrow$: Clear, by Theorem 2.6.
$\Rightarrow$: Now $L(\mathcal{A})$ is infinite, therefore it contains a word of the length at least n. Let w be (some) shortest word of this type. If $|w| < 2n$ we are done. Otherwise, by Theorem 2.6 we can factorize $w = uvz$ with $|uv| \leq n$ and $v \neq 1$, and moreover $uz \in L(\mathcal{A})$. But $|w| > |uz| \geq n$, a contradiction. □

Fundamental problems in formal language theory are different kinds of *decision problems*. In these problems it is asked whether a language of certain type (or more precisely a device determining the language) has a certain property. As a *solution* to a decision problem one has to construct an algorithm to solve the problem, or to prove that such an algorithm does not exist.

Standard decision problems are for example:

Membership problem , "$w \in L(\mathcal{A})$?" That is, given a word w and automaton $\mathcal{A}$, decide whether w is in $L(\mathcal{A})$.

Emptiness problem , "$L(\mathcal{A}) = \emptyset$?" That is, given an automaton $\mathcal{A}$, decide whether $L(\mathcal{A})$ contains no words (or equivalently a word).

Equivalence problem , "$L(\mathcal{A}) = L(\mathcal{B})$?" That is, given two automata, decide whether they accept the same language.

Remark 2.9. Of course, there is no reason to consider the above problems only for finite automata — $\mathcal{A}$ can be any device defining languages.

Remark 2.10. The above formulation of decision problems emphasizes the fact whether problems are in principal algorithmically decidable. Of course, in the case of decidable problems one can also ask, how complicated the problem is computationally. This aspect is only briefly considered in this course.

Theorem 2.8. *Membership, emptiness and equivalence problems are decidable for regular languages (given by FA´s accepting those).*

Proof. Membership problem: Here we are given a FA $\mathcal{A}$ and a word $w \in \Sigma^*$. An algorithm to test "$w \in L(\mathcal{A})$?" is obvious: Carry out the computations caused by w in $\mathcal{A}$, if some of them is accepting, output "yes", otherwise "no". In the case $\mathcal{A}$ is deterministic there exists only one such computation to be checked.

Emptiness problem: An algorithm: Decide the membership problem for all w satisfying $|w| <$ the number of states of $\mathcal{A}$, and if one of these is accepted then output "yes", otherwise "no". By Corollary 2.7 the algorithm is correct.

A more efficient algorithm is obtained as follows: Set

$$R_0 = \{q_0\}$$

and

$$R_{i+1} = R_i \cup \{q \mid \exists a \in \Sigma, p \in R_i : \ p \xrightarrow{a} q \text{ in } \mathcal{A}\}.$$

Compute R_i´s as long as they are properly increasing, and then test whether the set R_{i_0} such obtained contains a final state.

Clearly, the algorithm

- terminates, since the sequence of R_i´s is increasing and the total number of states is finite;
- works correctly, since R_i´s compute the states which are reachable by words of length at most i.

Equivalence problem: Here two FA´s $\mathcal{A}$ and $\mathcal{B}$ are given. The algorithm is based on the equivalence:

$$L(\mathcal{A}) = L(\mathcal{B}) \qquad \text{iff} \qquad \overbrace{\big(L(\mathcal{A}) \setminus L(\mathcal{B})\big) \cup \big(L(\mathcal{B}) \setminus L(\mathcal{A})\big)}^{=L} = \emptyset.$$

Clearly, this equivalence is correct. On the right hand side we have the emptiness problem for a certain regular language. It can be solved by our second problem, if we can algorithmically construct an FA for L. This, in turn, can be done by Theorem 2.3. □

Remark 2.11. Let us consider briefly the computational complexity of the above algorithms, i.e. how many steps are needed measured as the function on the size of an input. The size of a word is clear: its length. Let us measure the size of an automaton by the number of its states (more precise would be the number of transitions).

Assume further that the automaton is given as a DFA (if it would be an NFA then the size could be drastically smaller, cf. Remark 2.3 on page 12). Now the above problems can be solved as follows:
Membership: In time $\mathcal{O}(|w|)$, if the size of $\mathcal{A}$ is not counted and each computation step in $\mathcal{A}$ can be done in a constant time.

Emptiness: In time $\mathcal{O}(n|\Sigma|^n)$, where n is the size of $\mathcal{A}$, by the first algorithm; and in time $\mathcal{O}(n^2)$, by the second, assuming that the size of the alphabet of $\mathcal{A}$ is a constant.

Equivalence: In a polynomial time of some rather small degree. This follows from the second algorithm for the emptiness problem, and the fact that a deterministic automaton for L is of polynomial size in number of states of $\mathcal{A}$ and $\mathcal{B}$.

Finally, let us look what happens if we assume that the language is given by an NFA. For the membership problem the above trivial algorithm becomes exponential; for the emptiness the second algorithm remains polynomial of degree at most 3. The equivalence problem, in turn, becomes much more complicated (since the complementation may increase the number of states drastically). Indeed, for this problem no polynomial time algorithm is known, more precisely it is known to be so–called PSPACE–complete problem.

2.3 Characterizations

In this section we give several different characterizations for the family of regular languages. The first one is one of the oldest and most important results in automata theory.

Theorem 2.9 (Kleene, 1956). *A language $L \subseteq \Sigma^*$ is regular iff it is rational.*

Proof. $\Leftarrow$: We have to show that (i) the initial languages in the definition of rational languages, cf. page 5, are regular, and that (ii) the family of regular languages is closed under rational operations.

Condition (i) is clear: $\emptyset$ and $\{a\}$ are regular. Condition (ii), was proved in Theorem 2.4.

$\Rightarrow$: Let $L = L(\mathcal{A})$ for a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. We have to construct a rational expression for L. For $0 \leq m \leq n$ and $q_i, q_j \in Q$ let

$$L_{ij}^m = \left\{ w \in \Sigma^* \mid q_i \xrightarrow{w} q_j \text{ in } \mathcal{A} \quad \text{and} \quad \forall u < w, u \neq 1 : \ \delta(q_i, u) \in \{q_0, \ldots, q_{m-1}\} \right\}.$$

In other words, L_{ij}^m consists of exactly those words which lead in $\mathcal{A}$ from q_i to q_j using only intermediate states from the set $\{q_0, \ldots, q_{m-1}\}$. We shall show that these sets are rational, which implies the theorem, since

$$L = \bigcup_{q_j \in F} L_{0j}^n.$$

To prove that L_{ij}^m´s are rational we proceed by induction on m.

Case $m = 0$:
$$L_{ij}^0 = \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{, if } i \neq j, \\ \{a \in \Sigma \mid \delta(q_i, a) = q_j\} \cup \{1\} & \text{, if } i = j \end{cases}$$
proving that L_{ij}^0 is rational.

Induction step: We assume that languages L_{ij}^m for $0 \leq m < n$ and $0 \leq i, j \leq n-1$ are rational. And we claim that
$$L_{ij}^{m+1} = L_{ij}^m \cup L_{im}^m (L_{mm}^m)^* L_{mj}^m. \tag{2.3}$$
In order to prove this we conclude from the definition of L_{ij}^m´s that
$$w \in L_{ij}^{m+1} \quad \text{iff}$$

- w leads from q_i to q_j without visiting states q_k for $k > m$, iff
- w leads from q_i to q_j without visiting states q_k for $k > m-1$ *or*
- w leads from q_i to q_j visiting the state q_m, possibly several times, but without visiting the states q_k for $k > m$.

In the first case w belongs to the first member of the union of the right hand side of (2.3), and in the second case to the second member.

Formula (2.3) proves the claim by induction hypothesis. □

Remark 2.12. Again the proof of Theorem 2.9 is constructive. That is, given a regular expression one can construct an FA (and hence also a DFA) recognizing it, and conversely given a DFA one can construct a rational expression defining this language. The constructions are based on Theorem 2.4 and the proof of Theorem 2.9, respectively. The former problem is called the *synthesis problem* for finite automata and the latter the *analysis problem* for finite automata.

Our above solutions for analysis and synthesis problems are not computationally efficient. A more practical algorithm for the analysis problem can be based on the following:

Lemma 2.10. *Let $K \subseteq \Sigma^+$ and $L \subseteq \Sigma^*$ be regular languages. Then the equation $X = XK \cup L$ has a unique solution $X = LK^*$ which is regular.*

Proof. Clearly LK^* is a solution:
$$(LK^*)K \cup L = L(K^*K) \cup L = LK^+ \cup L = L(K^+ \cup \{1\}) = LK^*.$$
In order to prove the uniqueness let L_1 and L_2 be two different solutions, and w a minimal (with respect to length) word in the symmetric difference of L_1 and L_2, say in $L_1 \setminus L_2$. Now we can conclude:
$$w \notin L_2 \overset{L_2 = L_2K \cup L}{\Longrightarrow} w \notin L \overset{L_1 = L_1K \cup L}{\Longrightarrow} w \in L_1K,$$
so we can write $w = uv$, with $u \in L_1$ and $v \in K$. Moreover, since $K \subseteq \Sigma^+$, $|v| > 0$, so that $|u| < |w|$. But since $u \in L_1$, by the minimality of w, u must be in L_2, too. Hence we get a contradiction
$$w = uv \in L_2K \subseteq L_2.$$
□

Now, we apply Lemma 2.10 to the analysis problem.
Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA (or an NFA). Denote $q_0 = 1$,$Q = \{1, \dots, n\}$ and

$$\begin{array}{rll} L_i = & L(\mathcal{A}_i), & \text{for } \mathcal{A}_i = (Q, \Sigma, \delta, 1, \{i\}),\ i = 1, \dots, n, \\ K_{ij} = & \{a \in \Sigma \mid \delta(j, a) = i\}, & \text{for } 1 \le i, j \le n. \end{array}$$

These languages are connected by the identities:

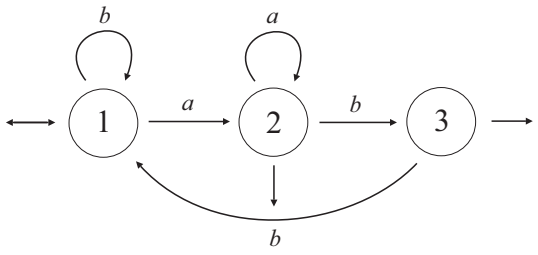
$$\begin{cases} L_1 = L_1K_{11} \cup L_2K_{12} \cup \cdots \cup L_nK_{1n} \cup \{1\} \\ L_2 = L_1K_{21} \cup L_2K_{22} \cup \cdots \cup L_nK_{2n} \\ \qquad\qquad \vdots \\ L_n = L_1K_{n1} \cup L_2K_{n2} \cup \cdots \cup L_nK_{nn}. \end{cases} \tag{2.4}$$

Here the languages L_i are unknown languages, while K_{ij}´s can be immediately determined from $\mathcal{A}$. Also we have

$$L = \bigcup_{i \in F} L_i.$$

Now, we apply Lemma 2.10 to solve (2.4). First since $K_{11} \subseteq \Sigma^+$ we can solve L_1, in terms of $L_2, \dots, L_n$, from the first equation and then substitute it to other equations. Thus a system with fewer unknowns is obtained, and the equations still satisfy the condition for the coefficients K. Consequently, the procedure can be continued, and finally the values for unknowns are found. Above applies for NFA´s as well.

Example 2.3. Consider the DFA $\mathcal{A}$:



We obtain (replacing $\cup$ by $+$):

$$\begin{cases} L_1 = L_1b + L_2\emptyset + L_3b + 1 \\ L_2 = L_1a + L_2a + L_3\emptyset \\ L_3 = L_1\emptyset + L_2b + L_3\emptyset \end{cases}$$

$$\Leftrightarrow \begin{cases} L_1 = L_1b + L_3b + 1 \\ L_2 = L_1a + L_2a \\ L_3 = L_2b \end{cases} \overset{L_3 = L_2b}{\Leftrightarrow} \begin{cases} L_1 = L_1b + L_2bb + 1 \\ L_2 = L_1a + L_2a \end{cases}$$

$$\overset{L_2 = L_1aa^*}{\Leftrightarrow} \quad L_1 = L_1b + L_1aa^*bb + 1 \ \Leftrightarrow\ L_1 = 1(b + a^+bb)^*$$

$$\therefore \quad \begin{cases} L_1 = (b + a^+bb)^* \\ L_2 = (b + a^+bb)^*a^+ \\ L_3 = (b + a^+bb)^*a^+b \end{cases} \quad \Rightarrow \quad \boxed{L(\mathcal{A}) = (b + a^+bb)^*\big(1 + a^+(1 + b)\big)}$$

In the second characterization of regular languages we use grammars. We recall that a context–free grammar $\mathcal{G} = (V, \Sigma, S, \mathcal{P})$ is *right linear* or *type* 3 iff the productions are of the form:

$$A \longrightarrow \alpha \quad \text{or} \quad A \longrightarrow \alpha B \qquad \text{with } A, B \in V \setminus \Sigma,\ \alpha \in \Sigma^*.$$

We call languages generated by such grammars *right linear* as well. Further we say that a right linear grammar is in a *normal form* if the productions are of the form:

$$A \longrightarrow aB, \quad A \longrightarrow a \quad \text{or} \quad A \longrightarrow 1 \qquad \text{with } A, B \in V \setminus \Sigma,\ a \in \Sigma.$$

Theorem 2.11. *$L \subseteq \Sigma^*$ is regular iff it is generated by a right linear grammar.*

Proof. $\Rightarrow$: Assume that $L = L(\mathcal{A})$ for a DFA $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$. Define a right linear grammar $\mathcal{G} = (V, \Sigma, q_0, \mathcal{P})$ by setting

$$\begin{aligned} V &= \Sigma \cup Q \quad \text{, where } \Sigma \cap Q \text{ is (assumed to be) empty,} \\ \mathcal{P} &= \{p \to aq \mid \delta(p,a) = q\} \cup \{p \to a \mid \delta(p,a) \in F\} \cup \mathcal{P}_0, \text{ where} \\ \mathcal{P}_0 &= \begin{cases} \{q_0 \to 1\} & \text{, if } q_0 \in F, \\ \emptyset & \text{, otherwise.} \end{cases} \end{aligned}$$

We claim that $L(\mathcal{A}) = L(\mathcal{G})$.

To prove this we first note that

$$1 \in L(\mathcal{G}) \quad \Leftrightarrow \quad q_0 \longrightarrow 1 \in \mathcal{P} \quad \Leftrightarrow \quad q_0 \in F \quad \Leftrightarrow \quad 1 \in L(\mathcal{A}).$$

Moreover, for $w \neq 1$ we have, by the construction,

$$p \xrightarrow{w} q \quad \text{in } \mathcal{A} \quad \Leftrightarrow \quad p \Rightarrow_{\mathcal{G}}^* wq$$

and

$$p \xrightarrow{w} q \in F \quad \text{in } \mathcal{A} \quad \Leftrightarrow \quad p \Rightarrow_{\mathcal{G}}^* w.$$

(If you want a precise formal proof for these equivalences, note that directions "$\Rightarrow$" are clear — paths in $\mathcal{A}$ give directly the derivations. For the reverse implication you can apply induction together with the fact, what you can say about the last steps of the derivation).

Since $S = q_0$ the claim follows.

$\Leftarrow$: Now assume that $L = L(\mathcal{G})$ for a right linear grammar $\mathcal{G} = (V, \Sigma, S, \mathcal{P})$. We define a GFA $\mathcal{A} = (Q, \Sigma, E, \{S\}, F)$ as follows:

$$\begin{aligned} Q &= (V \setminus \Sigma) \cup \{f\} \quad \text{, where } f \text{ is a new symbol,} \\ F &= \{f\} \quad \text{, and} \\ E &= \{(p, \alpha, q) \mid p \to \alpha q \in \mathcal{P},\ q \in V \setminus \Sigma\} \cup \{(p, \alpha, f) \mid p \to \alpha \in \mathcal{P},\ \alpha \in \Sigma^*\} \end{aligned}$$

As above we claim that $L(\mathcal{G}) = L(\mathcal{A})$.

To see this we first note that, if $w \in L(\mathcal{A})$, then $S \xrightarrow{w} f$ in $\mathcal{A}$, so that w possesses a factorization $w = u_1 \cdots u_n$, with $u_i \in \Sigma^*$, and moreover there exist states $q_1, \ldots, q_{n-1}$ such that

$$(S, u_1, q_1),\ (q_1, u_2, q_2),\ \ldots,\ (q_{n-2}, u_{n-1}, q_{n-1}),\ (q_{n-1}, u_n, f) \in E$$

and, by the construction, none of the q_i´s equal f. Consequently, $S \Rightarrow_{\mathcal{G}}^* w$.

Conversely, if $S \Rightarrow_{\mathcal{G}}^* w \in \Sigma^*$, then in the last step of the derivation a production of the form $p \to \alpha$, with $\alpha \in \Sigma^*$, must be applied, and in all other steps (if any) productions of the form $p \to \alpha q$, with $q \in V \setminus \Sigma$, must be applied. So it follows from the construction of $\mathcal{A}$ that $S \xrightarrow{w} f$ in $\mathcal{A}$, that is $w \in L(\mathcal{A})$. □

Corollary 2.12. *Each right linear language is generated by a right linear grammar in normal form.*

Proof. Let L be generated by a right linear grammar. Then, by Theorem 2.11, it is regular, and thus accepted by a DFA. But, by the first part of the proof of Theorem 2.11, a language accepted by a DFA, is generated by a right linear grammar in normal form. □

Remark 2.13. The construction indicated above to replace a right linear grammar with an equivalent normal form grammar is rather complicated:

$$\text{RLG} \longrightarrow \text{GFA} \longrightarrow \text{DFA} \longrightarrow \text{NRLG}.$$

We conclude our characterization results with some other algebraic characterizations. We need some terminology.

Definition 2.4. We recall that *equivalence relation* ρ on Σ^* is a relation, which is *reflexive*, *symmetric* and *transitive*, that is satisfies for all $u, v, w \in \Sigma^*$:

$$\begin{aligned} &u \ \rho \ u, \\ &u \ \rho \ v \quad \Rightarrow \quad v \ \rho \ u, \\ &u \ \rho \ v, \ v \ \rho \ w \quad \Rightarrow \quad u \ \rho \ w. \end{aligned}$$

Further an equivalence relation ρ is *right congruence* (resp. *congruence*) if it satisfies for all w (resp. w_1 and w_2)

$$\begin{aligned} &u \ \rho \ v \ \Rightarrow \ uw \ \rho \ vw \\ (\text{resp.} \quad &u \ \rho \ v \ \Rightarrow \ w_1uw_2 \ \rho \ w_1vw_2). \end{aligned}$$

Definition 2.5. Let $L \subseteq \Sigma^*$ be a language. We associate to L two equivalence relations $\sim_L$ and $\approx_L$ as follows:

$$u \sim_L v \qquad \text{iff} \qquad u^{-1}L = v^{-1}L \tag{2.5}$$

and so called *syntactic congruence* of L:

$$u \approx_L v \qquad \text{iff} \qquad \forall x, y \in \Sigma^* : \ [xuy \in L \ \Leftrightarrow \ xvy \in L]. \tag{2.6}$$

Since these relations are defined either by the equality or by the equivalence they are clearly equivalence relations. Moreover, $\sim_L$ is a right congruence:

$$\begin{aligned} u \sim_L v \ &\Rightarrow \quad u^{-1}L = v^{-1}L \quad \Rightarrow \ w^{-1}(u^{-1}L) = w^{-1}(v^{-1}L) \\ &\Rightarrow \ (uw)^{-1}L = (vw)^{-1}L \ \Rightarrow \quad uw \sim_L vw. \end{aligned}$$

The relation $\approx_L$, in turn, is a congruence (cf. Exercises).

Finally, we say that an equivalence relation is *finite* if the number of its equivalence classes is finite.

The above notions were associated with a language (which was not necessarily regular). Now, we associate a right congruence on Σ^* to a regular language L via a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ accepting L.

Definition 2.6. We define a relation $\sim_{\mathcal{A}}$ on Σ^* by the condition:

$$u \sim_{\mathcal{A}} v \qquad \text{iff} \qquad \delta(q_0, u) = \delta(q_0, v). \tag{2.7}$$

Again it is defined by the equality so that $\sim_{\mathcal{A}}$ is an equivalence relation, and since $\mathcal{A}$ is deterministic it is also a right congruence: If u and v leads in $\mathcal{A}$ to a same state, so do uw and vw, for any w.

Relations (2.5) and (2.7) are related as follows:

Lemma 2.13. *Let $L = L(\mathcal{A})$ for a DFA $\mathcal{A}$. Then for all $u, v \in \Sigma^*$:*

$$u \sim_{\mathcal{A}} v \quad \Rightarrow \quad u \sim_L v.$$

Proof. Assume that $u \sim_{\mathcal{A}} v$, that is with our standard notations $\delta(q_0, u) = \delta(q_0, v) = q$. We have to show that

$$u^{-1}L = v^{-1}L. \tag{2.8}$$

But, by the definition of the quotient,

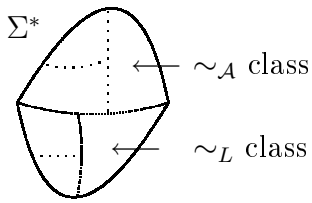
$$u^{-1}L = \{w \mid uw \in L\}$$

which, by the identity $L = L(\mathcal{A})$, is equal to the language

$$\{w \mid \delta(q_0, uw) \in F\} = \{w \mid \delta(q, w) \in F\}.$$

So (2.8) follows from the assumption. □

Lemma 2.13 says that the relation $\sim_{\mathcal{A}}$ is a *refinement* of $\sim_L$, that is each equivalence class of $\sim_L$ is a union of those of $\sim_{\mathcal{A}}$.



We prove

Theorem 2.14. *$L \subseteq \Sigma^*$ is regular iff $\sim_L$ is finite.*

Proof. $\Rightarrow$: Assume that $L = L(\mathcal{A})$ for a DFA $\mathcal{A}$. Since the number of states of $\mathcal{A}$ is finite, so is the right congruence $\sim_{\mathcal{A}}$, cf. (2.7). But then, by Lemma 2.13, $\sim_L$ is finite as well.

$\Leftarrow$: Assume that $\sim_L$ is finite, i.e. the set

$$Q = \{u^{-1}L \mid u \in \Sigma^*\}$$

is finite. We define a DFA $\mathcal{A}_L = (Q_L, \Sigma, \delta_L, i_L, F_L)$ as follows:

$$\begin{aligned} Q_L &= Q = \{u^{-1}L \mid u \in \Sigma^*\}, \\ i_L &= L = 1^{-1}L, \\ F_L &= \{u^{-1}L \mid u \in L\} \text{ and} \\ \delta_L(u^{-1}L, a) &= a^{-1}(u^{-1}L) = (ua)^{-1}L. \end{aligned}$$

Since Q is finite, this is a well defined DFA. Moreover,

$$\begin{aligned} w \in L(\mathcal{A}_L) &\Leftrightarrow \delta_L(L, w) \in F_L \Leftrightarrow w^{-1}L \in \{u^{-1}L \mid u \in L\} \\ &\Leftrightarrow \exists u \in L : \ w^{-1}L = u^{-1}L \Leftrightarrow w \in L, \end{aligned}$$

where the last equivalence is based on the fact $u \in L \Leftrightarrow 1 \in u^{-1}L$.

Consequently, we have found a DFA accepting L. □

Remark 2.14. The automaton $\mathcal{A}_L$ for the language L constructed above is called the *minimal* automaton for L. We justify this terminology later.

In the above proof we noted that

$$u \in L \quad \text{and} \quad w^{-1}L = u^{-1}L \qquad \Rightarrow \qquad w \in L,$$

that is, L is some union of equivalence classes of $\sim_L$ — and this is independent of whether L is regular or not. For regular L we can say more:

Theorem 2.15 (Myhill–Nerode). *A language $L \subseteq \Sigma^*$ is regular iff it is some union of equivalence classes of a finite right congruence on Σ^*.*

Proof. $\Rightarrow$: In the proof of Theorem 2.14: If L is regular, then $\sim_L$ is a finite right congruence and L is a union of some equivalence classes of $\sim_L$.

$\Leftarrow$: Assume that $L = \bigcup_{u \in F}\{w \mid w \ \rho \ u\}$, where ρ is a finite right congruence and F is a finite language. We claim, in accordance with Lemma 2.13, that for all $u, v \in \Sigma^*$

$$u \ \rho \ v \quad \Rightarrow \quad u \sim_L v. \tag{2.9}$$

So assume that $u \ \rho \ v$, that is, since ρ is a right congruence,

$$\forall w: \quad uw \ \rho \ vw. \tag{2.10}$$

Now, let $x \in u^{-1}L$. Then $ux \in L$ and so by (2.10) and the fact that L is a union of ρ-classes, also $vx \in L$, which means that $x \in v^{-1}L$. Consequently, by symmetry, $u \sim_L v$ and we have proved (2.9).

Finally, since ρ is finite, so is $\sim_L$, by (2.9), so that the regularity of L follows from Theorem 2.14. □

In Theorems 2.14 and 2.15 we characterized regular languages in terms of right congruences like $\sim_L$. Similarly, this family can be characterized by using the syntactic congruence $\approx_L$, which was defined by the formula (2.6). Intuitively, the formula means that u and v occur in words of L in exactly the same context.

Theorem 2.16. *A language $L \subseteq \Sigma^*$ is regular iff its syntactic congruence $\approx_L$ is finite.*

Proof. $\Leftarrow$: Assume that $\approx_L$ is finite. Then $\sim_L$ is finite, too, since as a congruence $\approx_L$ is also a right congruence, so that we can apply (2.9). Hence, by Theorem 2.15, L is regular.

$\Rightarrow$: Assume that L is regular, say it is accepted by a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. We associate each $w \in \Sigma^*$ with a partial mapping $t_w : Q \to Q$ by the condition:

$$t_w(q) = \delta(q, w).$$

Now, the relation defined by

$$u \ \rho \ v \qquad \text{iff} \qquad t_u = t_v$$

is a congruence: Since it is defined by the equality, it is an equivalence relation, and since $\mathcal{A}$ is deterministic it is a congruence. For example: for $x \in \Sigma^*$ if $t_u = t_v$, then

$t_{xu}(q) = \delta(q, xu) = \delta\big(\delta(q,x), u\big) = t_u\big(\delta(q,x)\big) = t_v\big(\delta(q,x)\big) = t_{xv}(q)$, so that $t_{xu} = t_{xv}$. Moreover, since the number of mappings t_w is finite, the index of ρ is finite, as well.

It follows that it is enough to prove that for all $u, v \in \Sigma^*$

$$u \;\rho\; v \quad \Rightarrow \quad u \approx_L v.$$

So assume that $u \;\rho\; v$, in other words, that $t_u = t_v$. Now for any $x, y \in \Sigma^*$ we have

$$\begin{array}{rcccl} xuy \in L & \Leftrightarrow & \delta(q_0, xuy) \in F & \Leftrightarrow & \delta\Big(\overbrace{\delta\big(\delta(q_0,x),u\big)}^{=t_u\left(\delta(q_0,x)\right)}, y\Big) \in F \\ & \Leftrightarrow & \delta\Big(\delta\big(\delta(q_0,x),v\big), y\Big) \in F & \Leftrightarrow & xvy \in L. \end{array}$$

□

Our final characterization result is in terms of monoids. We need some terminology.

Definition 2.7. We say that a monoid M *recognizes* $L \subseteq \Sigma^*$ if there exist a morphism $\varphi : \Sigma^* \to M$ and a subset $B \subseteq M$ such that $L = \varphi^{-1}(B)$. Languages recognized by finite monoids are often called *recognizable*.

Theorem 2.17. *A language $L \subseteq \Sigma^*$ is regular iff it is recognized by a finite monoid.*

Proof. $\Rightarrow$: Assume that L is regular. Then, by Theorem 2.16, the syntactic congruence $\approx_L$ is finite, and so is the quotient monoid $M = \Sigma^*/\!\approx_L$. Let $\varphi : \Sigma^* \to M$ be the canonical morphism: $\varphi(x) = [x]$. We claim that, in addition to these, we can take $B = \varphi(L)$. It remains to be shown that $L = \varphi^{-1}(B)$. But this is clear since L is a union of $\approx_L$-classes: $u \in L,\ u \approx_L v \ \Rightarrow\ [u \in L,\ 1 \cdot u \cdot 1 \in L \ \Leftrightarrow\ 1 \cdot v \cdot 1 \in L]$.

$\Leftarrow$: Let $\varphi : \Sigma^* \to M$, where M is a finite monoid, be a morphism and $B \subseteq M$ such that $L = \varphi^{-1}(B)$.

We define a DFA $\mathcal{A} = (M, \Sigma, \delta, 1, B)$, where

$$\delta(m, a) = m \cdot \varphi(a).$$

Clearly, $\mathcal{A}$ is well defined. Further, for any $w = a_1 \cdots a_n,\ a_i \in \Sigma$,

$$\delta(1, w) = 1 \cdot \varphi(a_1) \cdots \varphi(a_n) = \varphi(w),$$

so that indeed $L = \varphi^{-1}(B) = L(\mathcal{A})$. □

The monoid $M = \Sigma^*/\!\approx_L$ in the proof of Theorem 2.17 is called the *syntactic monoid* of the language L.

Remark 2.15. We have shown that the family of regular languages has many different characterizations. This is a clear evidence of the importance of the family. Further these characterizations are based on rather different notions. Two are based on the notion of accepting or generating words sequentially (FA and LRG), one is based on closure properties (rationality) and three more are based on finiteness of certain algebraic structures (right congruences, syntactic congruences and syntactic monoids). There exist still several other characterizations some of which are purely combinatorial.

2.4 Minimization

In this section we show that there exists the unique minimal (with respect to the number of states) *complete* DFA accepting a given regular language. This automaton is the automaton of the proof of Theorem 2.14, although in practice it is normally constructed by a different procedure.

Definition 2.8. Recall that *the minimal* automaton accepting a regular language $L \subseteq \Sigma^*$ was $\mathcal{A}_L = (Q_L, \Sigma, \delta_L, i_L, F_L)$, where

$$\begin{aligned} Q_L &= \{u^{-1}L \mid u \in \Sigma^*\}, \\ i_L &= L = 1^{-1}L, \\ F_L &= \{u^{-1}L \mid u \in L\} \quad \text{and} \\ \delta_L(u^{-1}L, a) &= a^{-1}(u^{-1}L) = (ua)^{-1}L. \end{aligned}$$

Clearly, $\mathcal{A}_L$ is deterministic, complete and as we saw $L(\mathcal{A}_L) = L$. Moreover, $\mathcal{A}_L$ is *connected*, that is each state is *reachable* from the initial one by some word u.

Now, let $\mathcal{A}$ be a complete, connected DFA accepting L. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, with $Q = \{q_0, \ldots, q_{n-1}\}$. Further let $L_i = L(\mathcal{A}_i)$, where $\mathcal{A}_i$ is obtained from $\mathcal{A}$ by taking q_i to its initial state. We define a mapping $\nu : Q \to Q_L$ by setting:

$$\nu(q) = u^{-1}L\ , \quad \text{where} \quad q = \delta(q_0, u).$$

We claim that

(i) ν is well defined, that is a mapping,

(ii) ν is surjective, and

(iii) if $\nu(q) = u^{-1}L$ and $\delta(q, a) = q'$, then $\nu(q') = (ua)^{-1}L$, in other words the following diagram commutes:

$$\begin{array}{ccccc} q & \xrightarrow{a} & q' & & \text{in } \mathcal{A} \\ {\scriptstyle\nu}\downarrow & & {\scriptstyle\nu}\downarrow & & \\ u^{-1}L & \xrightarrow{a} & (ua)^{-1}L & & \text{in } \mathcal{A}_L \end{array},$$

and moreover,

$$\nu(q_0) = i_L$$

and

$$q \in F \qquad \text{iff} \qquad \nu(q) \in F_L.$$

Proof of (i): Let $\delta(q_0, u) = q = \delta(q_0, v)$ for $u, v \in \Sigma^*$. We have to show that $u^{-1}L = v^{-1}L$. But this is just Lemma 2.13.

Proof of (ii): Clear, by the completeness of $\mathcal{A}$.

Proof of (iii): Let $\delta(q_0, u) = q$ and $\delta(q, a) = q'$. Then

$$\delta_L\big(\nu(q), a\big) = \delta_L(u^{-1}L, a) = (ua)^{-1}L = \nu(q') = \nu\big(\delta(q, a)\big)$$

proving the diagram. Secondly,

$$\nu(q_0) = 1^{-1}L = \text{ the initial state of } \mathcal{A}_L = i_L,$$

and finally,

$$q \in F \underset{\text{connected}}{\overset{\mathcal{A} \text{ is}}{\Leftrightarrow}} \exists u \in L:\ q_0 \xrightarrow{u} q \text{ in } \mathcal{A} \quad \Leftrightarrow \quad \exists u \in L:\ \nu(q) = u^{-1}L \quad \Leftrightarrow \quad \nu(q) \in F_L$$

It follows from (ii) that any complete DFA accepting L contains at least $n = |Q_L|$ states! Condition (iii), in turn, says that any complete DFA accepting L can be changed to $\mathcal{A}_L$ simply renaming the states by the function ν. In general, this renaming is not one–to–one, but if the DFA contains the minimal number of states, then it is one–to–one, that is a real renaming. So we obtain:

Theorem 2.18. *For each regular language L there exists the unique complete DFA of the minimal size obtainable from $\mathcal{A}_L$ by renaming the states.*

Of course, the above also gives a method to construct the minimal DFA: Compute all the sets $u^{-1}L$. This, however, is tedious, so we describe a more practical method.

Assume that a complete DFA $\mathcal{A}$ is found, for example, by the subset construction. We define the *minimization procedure*:

I. *Remove* from $\mathcal{A}$ those states (and transitions connected to them) which are not reachable from the initial state — this makes $\mathcal{A}$ connected. This can be done by a procedure similar to that computing 1-closure of a state in the proof of Theorem 2.2.

II. *Merge* two *equivalent* states of $\mathcal{A}$, that is states q_i and q_j satisfying

$$L_i = L_j. \tag{2.11}$$

Let us merge q_j to q_i. Then, of course, transitions leaving from q_j are removed, and transitions of the form $\delta(p, a) = q_j$ are replaced by $\delta(p, a) = q_i$. The automaton remains deterministic, and, by (2.11), equivalent to the original one. However, it does not need to be connected any more.

III. *Repeat* the above two procedures a finite number of times, until you find an automaton $\mathcal{A}_r$ (equivalent to $\mathcal{A}$), which is connected and *reduced*, that is does not contain two equivalent states.

By considerations proving Theorem 2.18, $\mathcal{A}_r$ can be renamed to $\mathcal{A}_L$. Moreover, since $\mathcal{A}_r$ is reduced, this renaming must be one–to–one. So we have proved

Theorem 2.19. *A complete DFA is the minimal one (up to a renaming the states) iff it is connected and reduced. Moreover, the minimization procedure always yields such a DFA.*

Remark 2.16. In the minimal DFA $\mathcal{A}_L$ there might be a state $u^{-1}L = \emptyset$. Of course, if this is the case it can be removed and a smaller DFA is found, but it is not anymore complete.

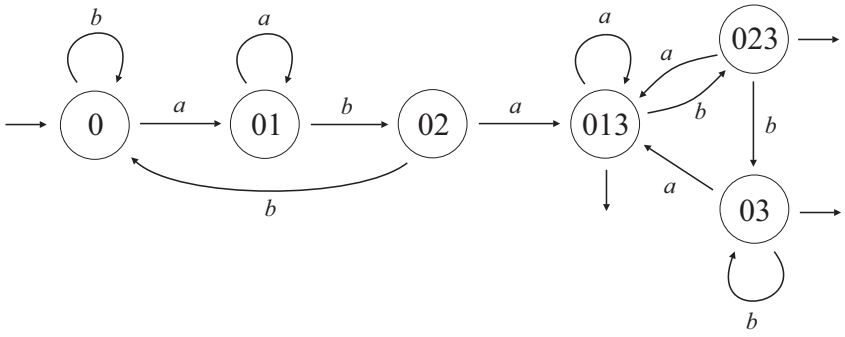
Remark 2.17. Assume that q_i and q_j are equivalent, i.e. they should be merged. Then also the states $\delta(q_i, w)$ and $\delta(q_j, w)$ are equivalent and forced to be merged (or deleted).

Remark 2.18. To apply the above minimization procedure, it is important to know how to test (2.11), i.e. $L_i \stackrel{?}{=} L_j$. This is a special case of the equivalence problem of DFA´s, and hence can be algorithmically solved, cf. page 19. A better algorithm can be based on the following equivalence:

$$L_i = L_j \qquad \text{iff} \qquad L_i \cap \Sigma^{<n} = L_j \cap \Sigma^{<n},$$

where $n = |Q|$ and $\Sigma^{<n}$ denotes the set of words shorter than n. We omit the proof here.

Example 2.1 (continued). We found on page 12 the following DFA for the language $\Sigma^* aba \Sigma^*$. Clearly, the states 013, 023 and 03 are equivalent, and can be merged, either directly or in two steps. All the other states are pairwise inequivalent:



$$1 \in L_{013} \setminus (L_0 \cup L_{01} \cup L_{02}) \,, \quad a \in L_{02} \setminus (L_0 \cup L_{01}) \quad \text{and} \quad ba \in L_{01} \setminus L_0.$$

2.5 Generalizations of FA

In the next two sections we define briefly three generalizations of FA, and consider a bit more FA with outputs, that is finite transducers.

Alternating finite automaton, AFA for short, is like an NFA, but acceptance is defined in a more general way. Recall that a DFA associates to an input w a unique computation, while an NFA associates to it a *computation tree*, which is accepting if at least one leaf is labeled by a final state. In an AFA the set of states is divided into two parts $Q_\exists$ and $Q_\forall$, *existential* and *universal* states, and an input word is accepted if any subtree starting from a universal state has the property that all leaves are labeled by accepting states. Without giving a formal definition we illustrate above as follows:

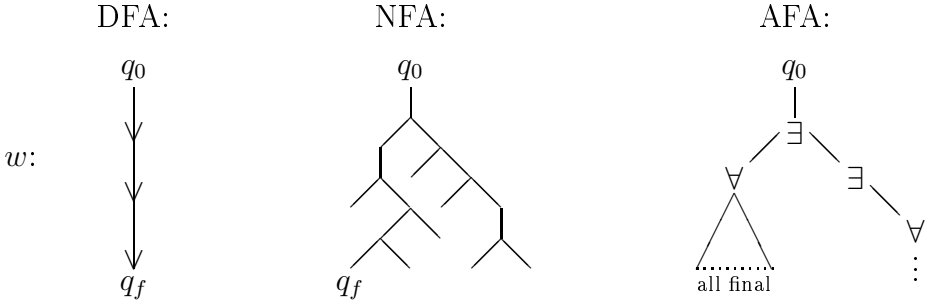


Figure 2.2: Accepting computations in DFA, NFA and AFA.

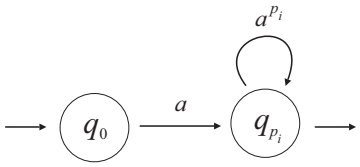
The following example shows that some languages can be accepted with a lot fewer states by an AFA than by an NFA.

Example 2.2. For a prime number p let $L_p = \{a^{np}a \mid n \geq 0\}$. Take the t first primes and consider the language

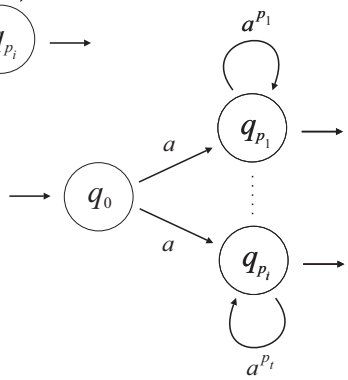
$$L_t = \bigcap_{i \leq t} L_{p_i} = \{a^{nq}a \mid n \geq 0\},$$

where p_i denotes the ith prime and $q = \prod_{i \leq t} p_i$. Clearly, any NFA accepting L_t must contain at least q states. On the other hand L_t is accepted by an AFA $\mathcal{A}$ constructed as follows:

First let $\mathcal{A}_{p_i}$ be a DFA accepting L_{p_i}:



Second $\mathcal{A}$ is built from these as shown beside:



Now, the number of states of $\mathcal{A}$ is $1+\sum_{i\le t} p_i = n$. It can be shown, by the result that there always exists a prime in the interval $[n, 2n]$, that q is not polynomially bounded on n.

On the other hand, one can show

Proposition 2.20. *Each language accepted by an AFA is regular.*

As another extension of an FA we mention *two–way finite automata*, 2FA for short. A (deterministic) two–way FA is $(Q, \Sigma, \delta, q_0, F)$, where δ is a partial mapping $Q \times \Sigma \to Q \times \{-1, 0, 1\}$ and everything else is as in DFA. The second component in the value of δ tells whether the reading head goes to the left, stay where it is, or goes to the right. The computation starts at the left end of the input word and is *accepting* if the automaton *enters to a final state after leaving the input word at its right end.*

Proposition 2.21. *Each language accepted by a 2FA is regular.*

Remark 2.19. Sometimes 2FA´s are defined with endmarkers, that is, input is of the form $\$w\#$, where $\$, \# \notin \Sigma$. Also one can define nondeterministic 2FA´s in a natural way. In both cases Proposition 2.21 holds.

Example 2.3. Let $L \subseteq \Sigma^*$ be accepted by an n-state DFA $\mathcal{A}$. Then $\$L^R\#$ is accepted by an $(n+2)$-state 2FA. Indeed, let $\mathcal{A} = (Q, \Sigma, \delta_1, i, F)$, and define the 2FA $\mathcal{A}_2$ as follows:

$$\begin{array}{rcll} \delta_2(q_0, a) & = & (q_0, 1) & \text{for } a \in \Sigma \cup \{\$\}, \\ \delta_2(q_0, \#) & = & (i, -1), & \\ \delta_2(q, a) & = & \big(\delta_1(q, a), -1\big) & \text{for } q \in Q,\ a \in \Sigma, \\ \delta_2(f, \$) & = & (q_f, 1) & \text{for } f \in F, \\ \delta_2(q_f, x) & = & (q_f, 1) & \text{for all } x, \end{array}$$

where q_0 and q_f are initial and final states of $\mathcal{A}_2$.

Now we apply the above to the following: Consider the language

$$L_i = \big\{w \in \{0,1\}^* \mid i\text{th letter equals to } 1\big\}.$$

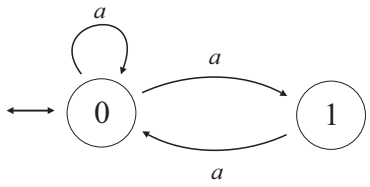
Clearly, L_i is accepted by an $(i+1)$-state DFA. Its reversal is accepted also by an $(i+1)$-state FA, but only nondeterministically. Indeed, one can show that any DFA accepting L_i^R contains at least 2^i states.

As a conclusion: 2FA may save exponential number of states!

As the third extension of an FA we consider *finite automata with multiplicities*, which leads to the *theory of rational power series*. Here we do not only specify, whether the input word is accepted or not, but also *count how many times it is accepted* — zero, of course, means that it is not accepted. Consequently, we associate with an NFA $\mathcal{A}$ a function $f_{\mathcal{A}} : \Sigma^* \to \mathbb{N}$

$$f_{\mathcal{A}}(w) = \text{ the number of times } w \text{ is accepted in } \mathcal{A}.$$

Example 2.4. Consider the following NFA $\mathcal{A}$. Then clearly, $f_{\mathcal{A}}(1) = 1$, $f_{\mathcal{A}}(a) = 1$ and



$$f_{\mathcal{A}}(a^n) = f_{\mathcal{A}}(a^{n-1}) + f_{\mathcal{A}}(a^{n-2}), \quad \text{for } n \geq 2,$$

since each accepting path is uniquely of the form $0 \xrightarrow{a^{n-1}} 0 \xrightarrow{a} 0$ or of the form $0 \xrightarrow{a^{n-2}} 0 \xrightarrow{a} 1 \xrightarrow{a} 0$. Consequently, the value of $f_{\mathcal{A}}$ on a^n is the nth Fibonacci number.

As an example of results in this theory we state

Proposition 2.22. *Given two NFA $\mathcal{A}_1$ and $\mathcal{A}_2$. It is undecidable whether*

$$f_{\mathcal{A}_1}(w) \leq f_{\mathcal{A}_2}(w) \quad \textit{for all } w \in \Sigma^*.$$

Remark 2.20. Recently the above theory has turned out useful in computer graphics, in generating and compressing pictures.

2.6 Finite transducers

This section is devoted to *finite transducers*, which are finite automata with outputs. So far the only output we have had has been "accept" or "reject". Finite transducers are capable of producing outputs in every step of their computations, thus computing functions (or relations) $\Sigma^* \to \Delta^*$.

Definition 2.9. A *finite transducer*, FT for short, is a sixtuple $\mathcal{T} = (Q, \Sigma, \Delta, E, q_0, F)$, where

Q is a finite *set of states*,

Σ and Δ are *input and output alphabets*,

$E \subseteq Q \times \Sigma^* \times \Delta^* \times Q$ is a *finite set of transitions*,

$q_0 \in Q$ is the *initial state*,

$F \subseteq Q$ is the set of *final states*.

If we forget the output structure, that is Δ, we obtain a GFA, the *underlying FA of* $\mathcal{T}$. If the underlying FA is an NFA, then $\mathcal{T}$ is called a *generalized sequential machine*, gsm for short, or a *sequential transducer*. Finally, if the underlying FA is a DFA, then $\mathcal{T}$ is called a *deterministic generalized sequential machine*, dgsm for short (actually, sometimes gsm and dgsm are defined without final states). By a *normalized* FT we mean a FT satisfying $E \subseteq Q \times (\Sigma \cup \{1\}) \times (\Delta \cup \{1\}) \times Q$.

We can extend our notations of an FA in a straightforward way:

$$p \xrightarrow{u,v} q \qquad \text{means that } (p,u,v,q) \in E;$$

$$p \xrightarrow{u,v} q \text{ in } \mathcal{T} \quad \begin{cases} \text{means that there exist } n \geq 0, \text{ words } u_1,\ldots,u_n \text{ in } \Sigma^*, \\ v_1,\ldots,v_n \text{ in } \Delta^*, \text{ and states } p_0,\ldots,p_n \text{ such that } p = \\ p_0,\ q = p_n \text{ and } (p_{i-1},u_i,v_i,p_i) \in E \text{ for } i = 1,\ldots,n. \end{cases}$$

Now a finite transducer *defines the relation* $R(\mathcal{T}) : \Sigma^* \to \Delta^*$ or *computes the function* or *transduction* $\mathcal{T} : \Sigma^* \to \mathcal{P}(\Delta^*)$

$$R(\mathcal{T}) = \{(u,v) \in \Sigma^* \times \Delta^* \mid \exists q \in F :\ q_0 \xrightarrow{u,v} q \text{ in } \mathcal{T}\}.$$

Here $\mathcal{P}(\Delta^*)$ denotes the power set of Δ^*, i.e. the family of subsets of Δ^*.

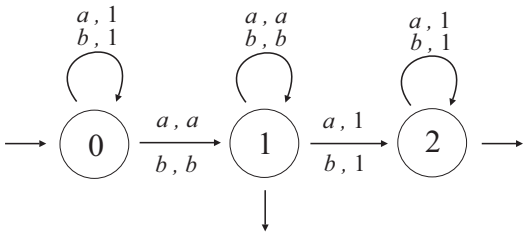
The *domain* and *range of* $\mathcal{T}$ are defined in a natural way:

$$\mathrm{dom}(\mathcal{T}) = \{u \in \Sigma^* \mid \exists v \in \Delta^* :\ (u,v) \in R(\mathcal{T})\},$$
$$\mathrm{range}(\mathcal{T}) = \{v \in \Delta^* \mid \exists u \in \Sigma^* :\ (u,v) \in R(\mathcal{T})\}.$$

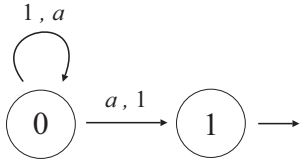
Finally, a relation R of $\Sigma^* \times \Delta^*$, often denoted $R : \Sigma^* \to \Delta^*$, is *rational* if it is defined by a finite transducer. A *rational function* $\Sigma^* \to \Delta^*$ is a rational relation, which is a *partial* function.

Remark 2.21. Finite transducers are special cases of finite automata on arbitrary monoids. Indeed, the labels of the transitions are elements of the product monoid $\Sigma^* \times \Delta^*$. But, since this monoid is not free, the theory is in many respects more complicated.

Example 2.5. The following FT computes all nonempty factors of any input word $w \in \{a,b\}^*$.



Example 2.6. Note that the value of $\mathcal{T}$ for a given input w need not be finite, in general:



Directly from the definitions above we derive:

Theorem 2.23. *For each FT both* $\mathrm{dom}(\mathcal{T})$ *and* $\mathrm{range}(\mathcal{T})$ *are regular.*

It is also clear, cf. Examples 2.5 and 2.6, that we cannot eliminate the empty word from the labels of transitions. However, the easy part of the proof of Theorem 2.2 immediately yields:

Theorem 2.24. *For each FT there exists an equivalent normalized FT.*

Example 2.7. The partial function $f : \{a,b\}^* \to \{a,b\}^*$

$$\begin{aligned} f\big((ab)^n\big) &= a^n b^n \quad \text{for } n \geq 0, \\ f(w) &= 1 \qquad \text{for } w \notin (ab)^* \end{aligned}$$

is not rational. Indeed, if it were, then by Theorem 2.23 $\mathrm{range}(f) = \{a^n b^n \mid n \geq 0\}$ would be regular, which is not the case.

Our next theorem shows a connection between rational relations and rational (regular) languages.

Theorem 2.25 (Nivat, 1968). *A relation $R \subseteq \Sigma^* \times \Delta^*$ is rational iff there exist an alphabet Γ, morphisms $h : \Gamma^* \to \Sigma^*$ and $g : \Gamma^* \to \Delta^*$ and a regular language $L \subseteq \Gamma^*$ such that*

$$R = \big\{\big(h(w), g(w)\big) \mid w \in L\big\}. \tag{2.12}$$

Proof. $\Rightarrow$: Let $R = R(\mathcal{T})$ for a FT $\mathcal{T} = (Q, \Sigma, \Delta, E, q_0, F)$. Set $\Gamma = E$ and define

$$\begin{aligned} I &= \{(p,u,v,q) \in E \mid p = q_0\} \subseteq \Gamma, \\ T &= \{(p,u,v,q) \in E \mid q \in F\} \subseteq \Gamma, \\ M &= \{(p,u,v,q)(p',u',v',q') \in E^2 \mid q \neq p'\} \subseteq \Gamma^2, \end{aligned}$$

and finally,

$$L = (I\,\Gamma^* \cap \Gamma^*\,T) \setminus \Gamma^* M\,\Gamma^*.$$

Consequently, by the closure properties of Reg, L is regular. We shall show

Claim. L consists of exactly those sequences of transitions of $\mathcal{T}$, which are accepting in $\mathcal{T}$.

Now, by the definition of I, T and M, sequence s is in L iff

- it starts with a symbol in I, that is, with a transition having the first component equal to q_0, *and*
- it ends with a symbol in T, that is, with a transition having the last component in F, *and*
- it does not contain a factor from M, that is, consecutive transitions for which the last component of the former $\neq$ the first component of the latter.

The last condition is clearly equivalent to: in each factor of s of the length 2 the last component of the former letter = the first component of the latter letter.

So the claim follows.

Now, the presentation (2.12) follows when we define the morphisms

$$\begin{aligned} h &: \Gamma^* \to \Sigma^*, \quad h(p,u,v,q) = u \\ g &: \Gamma^* \to \Delta^*, \quad g(p,u,v,q) = v \end{aligned}$$

and

for all $(p,u,v,q) \in \Gamma$.

$\Leftarrow$: Assume that R has the presentation (2.12) with $L = L(\mathcal{A})$ for a DFA $\mathcal{A} = (Q, \Gamma, \delta, q_0, F)$. Change $\mathcal{A}$ to a FT $\mathcal{T} = (Q, \Sigma, \Delta, E, q_0, F)$ by setting

$$p \xrightarrow{h(a),\,g(a)} q \ \text{ in } E \qquad \text{iff} \qquad \delta(p,a) = q.$$

Obviously, $R(\mathcal{T}) = R$. □

Theorem 2.25 has a number of variations or corollaries. To state those we recall some terminology. First a relation $R \subseteq \Sigma^* \times \Delta^*$ can be viewed as a many valued function $R : \Sigma^* \to \Delta^*$, so that, for example, the expression "$(u, v) \in R$" can be written as "$v \in R(u)$". Second a composition of relations $R : \Sigma^* \to \Delta^*$ and $R' : \Delta^* \to \Gamma^*$ is well defined: $(u, v) \in R \circ R' : \Sigma^* \to \Gamma^*$ iff $\exists w \in \Delta^* : (u, w) \in R$ and $(w, v) \in R'$. Further we need the operation "*intersection with a regular language*": For a given regular language $L \subseteq \Sigma^*$ the operation $\bigcap L : \Sigma^* \to \Sigma^*$ maps w to the language $\{w\} \cap L$.

Corollary 2.26. *A relation $R : \Sigma^* \to \Delta^*$ is rational iff it can be factorized as*

$$R = g \circ \bigcap L \circ h^{-1},$$

where $L \subseteq \Gamma^*$ is regular and $h : \Gamma^* \to \Sigma^*$, $g : \Gamma^* \to \Delta^*$ are morphisms.

Proof. This is just a reformulation of Theorem 2.25:

$$\begin{array}{rcccl}
(u, v) \in R & \Leftrightarrow & v \in R(u) & \Leftrightarrow & v \in g \circ \bigcap L \circ h^{-1}(u) \\
& \Leftrightarrow & v \in g\big(h^{-1}(u) \cap L\big) & \Leftrightarrow & \exists w \in L :\ v = g(w), w \in h^{-1}(u) \\
& \Leftrightarrow & \exists w \in L :\ v = g(w), u = h(w) & \Leftrightarrow & (u, v) \in \text{r.h.s. of (2.12)}.
\end{array}$$

□

Corollary 2.27. *For each FT $\mathcal{T}$ and a regular language L $\mathcal{T}(L)$ is regular, that is the family of regular languages is closed under rational transductions.*

Proof. We have to show that $\mathcal{T}(L) = \{\mathcal{T}(w) \mid w \in L\}$ is regular. But, by Corollary 2.26, $\mathcal{T}(L) = g\big(h^{-1}(L) \cap L'\big)$, for suitably defined morphisms h and g, and a regular language L'. So the claim follows from closure properties of the family Reg. □

Example 2.7 (continued). There exists no FT $\mathcal{T}$ satisfying $\mathcal{T}\big((ab)^n\big) = a^n b^n$. Indeed, if such an FT would exist then $\mathcal{T}\big((ab)^*\big) = \{a^n b^n \mid n \geq 0\}$ would be regular.

Before stating a sharpening of Theorem 2.25, we need one more notion.

Definition 2.10. A morphism $\pi : \Sigma^* \to \Delta^*$, with $\Delta \subseteq \Sigma$, is a *projection* if $\pi(a) = a$, for $a \in \Delta$, and $\pi(a) = 1$, for $a \in \Sigma \setminus \Delta$.

Corollary 2.28. *Each rational relation $R \subseteq \Sigma^* \times \Delta^*$, with $\Sigma \cap \Delta = \emptyset$, has a presentation*

$$R = \big\{\big(\pi_1(w), \pi_2(w)\big) \mid w \in L_1\big\},$$

where $L_1 \subseteq (\Sigma \cup \Delta)^*$ is regular and $\pi_1 : (\Sigma \cup \Delta)^* \to \Sigma^*$ and $\pi_2 : (\Sigma \cup \Delta)^* \to \Delta^*$ are projections.

Proof. Follows directly from the proof of Theorem 2.25, when we assume that transducer $\mathcal{T}$ is normalized, as we can do by Theorem 2.24, and set

$$L_1 = f(L),$$

where $L \subseteq \Gamma^*$ is as in the proof of Theorem 2.25, and $f : \Gamma^* \to (\Sigma \cup \Delta)^*$ is a morphism defined as:

$$\begin{array}{ll}
f(p, a, 1, q) = a & \text{for all } a \in \Sigma,\ p, q \in Q, \\
f(p, 1, b, q) = b & \text{for all } b \in \Delta,\ p, q \in Q, \\
f(p, 1, 1, q) = 1 & \text{for all } p, q \in Q.
\end{array}$$

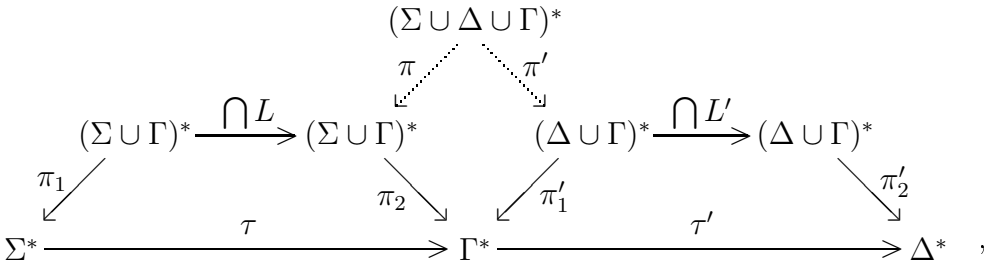
and

□

Now we are ready for another basic result of rational relations.

Theorem 2.29. *Rational relations are closed under composition.*

Proof. Let $\mathcal{T} : \Sigma^* \to \Gamma^*$ and $\mathcal{T}' : \Gamma^* \to \Delta^*$ be rational relations. By renaming we can assume that $\Sigma \cap \Gamma = \emptyset$ and $\Delta \cap \Gamma = \emptyset$. Hence we can apply Corollary 2.28, and so by Corollary 2.27, we have the situation illustrated by the solid lines of the following diagram:



where L and L' are regular languages over appropriate alphabets and all π´s are projections of the indicated form.

Assume first that $\Sigma \cap \Delta = \emptyset$. Add to the above diagram the projections π and π' (denoted by dotted lines). We claim that

$$(\pi'_1)^{-1} \circ \pi_2 = \pi' \circ \pi^{-1}. \tag{2.13}$$

Indeed, for any $w = \alpha_0 a_1 \alpha_1 \cdots a_n \alpha_n$, with $\alpha_i \in \Sigma^*$, $a_i \in \Gamma$, we have

$$(\pi'_1)^{-1} \circ \pi_2(w) = \{\beta_0 a_1 \cdots a_n \beta_n \mid \beta_1, \ldots, \beta_n \in \Delta^*\} = \pi' \circ \pi^{-1}(w).$$

Now, we can write

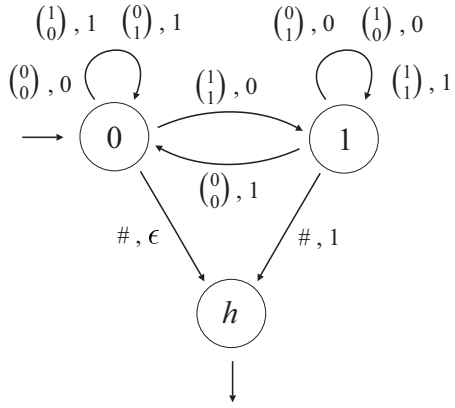
$$\begin{aligned}
(\mathcal{T}' \circ \mathcal{T})(u) &= \pi'_2\Big((\pi'_1)^{-1}\Big(\pi_2\big(\pi_1^{-1}(u) \cap L\big)\Big) \cap L'\Big) \\
&= \pi'_2\Big((\pi'_1)^{-1} \circ \pi_2\big(\pi_1^{-1}(u) \cap L\big) \cap L'\Big) \\
&\overset{(2.13)}{=} \pi'_2\Big(\pi' \circ \pi^{-1}\big(\pi_1^{-1}(u) \cap L\big) \cap L'\Big) \\
&\overset{(*)}{=} \pi'_2\Big(\pi'\big((\pi_1 \circ \pi)^{-1}(u) \cap \pi^{-1}(L)\big) \cap L'\Big) \\
&\overset{(**)}{=} \pi'_2 \circ \pi'\big((\pi_1 \circ \pi)^{-1}(u) \cap \pi^{-1}(L) \cap (\pi')^{-1}(L')\big) \\
&= h\big(g^{-1}(u) \cap L''\big),
\end{aligned}$$

where $h = \pi'_2 \circ \pi'$, $g = \pi_1 \circ \pi$ are morphisms and $L'' = \pi^{-1}(L) \cap (\pi')^{-1}(L')$ is a regular language. Step $(*)$ follows since inverse mappings are distributive over intersections, and $(**)$ follows from the identity $\pi(K) \cap L = \pi\big(K \cap \pi^{-1}(L)\big)$, which holds when π is a projection.

So we conclude, by Theorem 2.25, that $\mathcal{T} \circ \mathcal{T}'$ is rational if $\Sigma \cap \Delta = \emptyset$. If this is not the case, we rename Δ by a morphism ν, apply the above, and finally remove the renaming by ν^{-1}. □

We conclude this section with two examples.

Example 2.8. We show that an FT can add two binary numbers. We use the reverse binary presentation for binary numbers. The input for an FT is a pair presenting the same bit of numbers, and the machine uses an endmarker. The transducer is given by its graph, where for clarity we use ϵ for the empty word.



Example 2.9. Contrary to the above we claim that no FT can compute the product of binary numbers (under the above notations). Assume the contrary: $\mathcal{T}_\times$ computes the product. Consider the word

$$w_n = 10^{n-1}1, \quad n \geq 2,$$

which represents the number $2^n + 1$. So by our assumption,

$$\mathcal{T}_\times(\tbinom{w_n}{w_n}\#) = 10^n 10^{n-2}1.$$

Clearly, the language

$$L = \{\tbinom{w_n}{w_n}\# \mid n \geq 1\} = \tbinom{1}{1}\tbinom{0}{0}^+\tbinom{1}{1}\# \subseteq \{\tbinom{0}{0}, \tbinom{1}{1}, \#\}^*$$

is regular. So should be $\mathcal{T}_\times(L)$, by Corollary 2.27, but this is not the case.

Chapter 3

Context–free languages

3.1 Context–free grammars

Context–free languages have turned out very useful in different areas of computer science, such as in the theory of programming languages, in compiling and in parsing. Context–free languages, *CF languages* for short, as well as context–free grammars, *CF grammars* for short, were defined on pages 6 and 7. In order to fix notations we recall that a CF grammar is a quadruple

$$\mathcal{G} = (V, \Sigma, \mathcal{P}, S),$$

where $\mathcal{P}$ consists of the productions of the form

$$A \longrightarrow \beta\ , \quad \text{with } A \in N = V \setminus \Sigma \text{ and } \beta \in V^*.$$

Productions of this form are called *CF productions*, in general. CF productions are *linear*, if they are of the form

$$A \longrightarrow \alpha B \alpha'\ , \quad \text{with } A, B \in N \text{ and } \alpha, \alpha' \in \Sigma^*.$$

CF grammar is called *linear*, if its productions are linear, and the family of languages generated by such grammars is called the family of *linear languages*, Lin for short.

Example 3.1. Consider the grammar $\mathcal{G}$ with the productions

$$S \longrightarrow SS \mid (S) \mid 1,$$

where $\Sigma = \{\,(\,,\,)\,\}$. We claim that $L(\mathcal{G})$ consists of exactly all correctly built sequences of right and left parenthesis. Indeed, it follows from the productions (by induction) that all w in $L(\mathcal{G})$ are in this language. Conversely, again by induction, each such sequence can be generated by above productions, cf. Exercises. This language is denoted by D_1 and called *Dyck language* over the binary alphabet.

Example 3.2. The CF grammar with productions

$$\begin{aligned} E &\longrightarrow E + T \mid T, \\ T &\longrightarrow T * F \mid F, \\ F &\longrightarrow (E) \mid a, \end{aligned}$$

and $\Sigma = \{a, (,), +, *\}$, and E as the start symbol generates all correct arithmetic expressions, as can be seen straightforwardly.

Next, we fix some further terminology.

Definition 3.1. Consider a derivation according to $\mathcal{G}$:

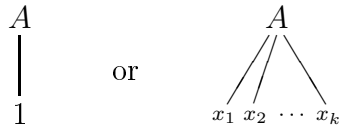
$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_{n-1} \Rightarrow w. \tag{D}$$

The derivation (D) is called *terminal*, if $w \in \Sigma^*$. Words w_i, as well as w, are called *sentential forms*. If in (D), in each derivation step the nonterminal rewritten in w_i is the leftmost one, then the derivation is called the *leftmost* derivation of w. Of course, we can consider also derivations starting from $A \in N \setminus \{S\}$ — these are called *A-derivations*.

Now we associate the derivation (D) (or A-derivation in general) with a *derivation tree* (or *A-derivation tree*) as follows:

(i) For $n = 1$:

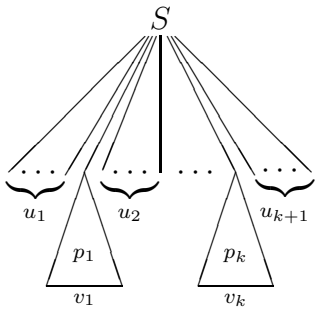
$A \Rightarrow 1$ or $A \Rightarrow x_1 \cdots x_k,\ x_i \in V$:



(ii) For $n \geq 2$:

If $w_1 = u_1 A_1 u_2 \cdots A_k u_{k+1}$ and
$w = u_1 v_1 u_2 \cdots v_k u_{k+1}$ and
$A_i \Rightarrow^* v_i$ is associated with p_i: 

then (D) is associated with the tree shown besides.



Of course, the correspondence between derivations and derivation trees is not one–to–one, in general. However, if we restrict our considerations to the leftmost derivations only, then this correspondence becomes one–to–one.

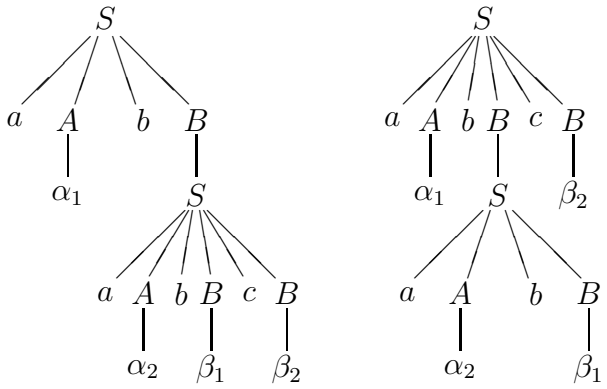
From a derivation tree T we can read the word generated by a corresponding derivation simply by catenating the leaves from left to right. This word is called the *yield* of T, in symbols yield(T). Finally, $T(\mathcal{G})$ denotes the set of all trees associated with terminal derivations, so that $L(\mathcal{G}) = \text{yield}\big(T(\mathcal{G})\big)$.

Definition 3.2. We call CF grammar $\mathcal{G}$ *ambiguous*, if for some word $w \in L(\mathcal{G})$ there exist at least two different derivation trees, that is two different leftmost derivations (of course, derivations are the same if they consist of exactly the same sequence of applications of productions). Otherwise $\mathcal{G}$ is *unambiguous*. A CF language is *inherently ambiguous*, if each grammar generating it is ambiguous.

Example 3.3. A CF grammar with the productions

$$\begin{aligned} S &\longrightarrow aAbB \mid aAbBcB, \\ A &\longrightarrow \alpha_1 \mid \alpha_2, \\ B &\longrightarrow S \mid \beta_1 \mid \beta_2 \end{aligned}$$

is ambiguous:



An example of a CF language which can be proved inherently ambiguous is the language $\{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$.

We move to prove several *normal forms* for CF grammars.

Definition 3.3. Let $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ be a CF grammar. We call a nonterminal A

– *terminating*, if $\{w \in \Sigma^* \mid A \Rightarrow_{\mathcal{G}}^* w\} \neq \emptyset$, and

– *reachable*, if $\exists u, v \in V^* : \ S \Rightarrow_{\mathcal{G}}^* uAv$.

Further $\mathcal{G}$ is called *reduced*, if all of its nonterminals are both terminating and reachable, or formally

$$\forall A \in V, \ \exists u, v \in V^*, w \in \Sigma^* : \quad S \ \Rightarrow_{\mathcal{G}}^* \ uAv \ \Rightarrow_{\mathcal{G}}^* \ uwv.$$

Theorem 3.1. *For each CF grammar one can effectively find an equivalent reduced CF grammar.*

Proof. Let $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. We construct the set T of *terminating* nonterminals and the set R of *reachable* nonterminals as follows:

$$\begin{aligned} T_0 &= \{A \in N \mid \exists w \in \Sigma^* : \ A \to w \in \mathcal{P}\}, \\ T_{i+1} &= T_i \cup \{A \in N \mid \exists u \in (T_i \cup \Sigma)^* : \ A \to u \in \mathcal{P}\}, \quad \text{for } i \geq 0, \end{aligned}$$

and

$$\begin{aligned} R_0 &= \{S\}, \\ R_{i+1} &= R_i \cup \{A \in N \mid \exists B \in R_i, \ u, v \in V^* : \ B \to uAv \in \mathcal{P}\}, \quad \text{for } i \geq 0. \end{aligned}$$

We note that:

1) $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_i \subseteq \cdots$,

2) If $T_i = T_{i+1}$, then also $T_{i+1} = T_{i+2}$,

3) $T_i = T_{i+1}$ at the latest when $i = |N| - 1$, and

4) $T = \bigcup_{i \geq 0} T_i$.

Consequently, by 1)–3) T can be effectively found.

The argument for finding R is exactly the same.

Now we construct an equivalent reduced CF grammar for $\mathcal{G}$. This is done in two steps.

I. *Eliminate all nonterminating nonterminals.* By the above procedure, we can find the set $N \setminus T$, and omit these and productions using these nonterminals (in either of the sides). We claim that the grammar thus obtained, say $\mathcal{G}_T$, is equivalent to $\mathcal{G}$.

First $\mathcal{G}_T$ cannot generate any terminal words outside $L(\mathcal{G})$, since all productions of $\mathcal{G}_T$ are available in $\mathcal{G}$. Conversely, if $w \in L(\mathcal{G})$, w is in Σ^* and it has a derivation D_w according to $\mathcal{G}$. Since $w \in \Sigma^*$ all nonterminals in D_w are in T, so that D_w is a derivation according to $\mathcal{G}_T$, as well.

II. *Eliminate all nonreachable nonterminals in $\mathcal{G}_T$.* As in case I this is done by throwing out some nonterminals and productions associated to those — namely those which are nonreachable in $\mathcal{G}_T$. A new grammar thus obtained, say $\mathcal{G}_{TR}$, is equivalent to $\mathcal{G}_T$ (and hence also to $\mathcal{G}$). Indeed, as above the inclusion $L(\mathcal{G}_{TR}) \subseteq L(\mathcal{G}_T)$ is clear, and

the reverse follows from the fact that any (terminating) derivation of $\mathcal{G}_T$ contains only reachable nonterminals, and so it is a derivation of $\mathcal{G}_{TR}$, too.

To conclude the proof, we claim that $\mathcal{G}_{TR}$ is reduced. Surely it contains only reachable nonterminals. So it is enough to show that any nonterminal of $\mathcal{G}_{TR}$ is terminating in $\mathcal{G}_{TR}$. Assume the contrary, that A is not terminating in $\mathcal{G}_{TR}$. Now, A is terminating in $\mathcal{G}_T$, but not in $\mathcal{G}_{TR}$, which means that for some B we have

$$A \Rightarrow^*_{\mathcal{G}_T} uBv \Rightarrow^*_{\mathcal{G}_T} w \in \Sigma^*,$$

where A is in $\mathcal{G}_{TR}$, but B *is not*. Since A is nonterminal of $\mathcal{G}_{TR}$, it is reachable in $\mathcal{G}_T$. But then, by above, so is B, and thus $B \in \mathcal{G}_{TR}$, a contradiction. □

Remark 3.1. The two steps of the proof of Theorem 3.1 cannot be done in the other order: Assume that $A \to BC$ is the only production for A and

$$S \Rightarrow^* \cdots A \cdots \Rightarrow^* \cdots BC \cdots$$

and

$$B \Rightarrow^* w \in \Sigma^* \quad \text{and} \quad \forall u: \text{ if } C \Rightarrow^* u, \text{ then } u \notin \Sigma^*.$$

Now, if we first remove nonreachable terminals and then nonterminating ones, then A, B and C remain in the first step, and in the second step C and A are removed, but B is not. However, B does not remain reachable!

In the above normal form we eliminated unnecessary nonterminals. In what follows, we restrict the form of productions in different ways.

Definition 3.4. We say that a CF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ is in

A) *Chomsky normal form* if the productions are of the form

(i) $A \to BC,\ A, B, C \in N$,

(ii) $A \to a,\ A \in N, a \in \Sigma$, or

(iii) $S \to 1$,

and moreover, if the production $S \to 1$ occurs, then $B \neq S$ and $C \neq S$ for all productions in (i).

B) *Greibach normal form* if the productions are of the form

(i) $A \to aA_1 \dots A_n,\ n \geq 0,\ A, A_i \in N, a \in \Sigma$, or

(ii) $S \to 1$, and in this case, S does not occur on the r.h.s. in other productions.

C) *Standard* 2*-form* if it is in Greibach normal form with n at most 2 for all productions in (i) of B).

Theorem 3.2. *Each CF language is generated by a CF grammar $\mathcal{G}_{CNF}$ in Chomsky normal form. Moreover, $\mathcal{G}_{CNF}$ can be effectively found.*

Proof. Let L be generated by a CF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. We first replace S by a new initial symbol S' and add the production $S' \to S$. Clearly, the language generated is not changed, and S' does not occur on the right hand sides of the productions. By renaming we can assume that $\mathcal{G}$ satisfies this condition.

Next we eliminate from $\mathcal{G}$ productions of the form $B \to 1$, with $B \neq S$. Define a subset N_1 of N by:

$$N_1 = \{B \in N \setminus \{S\} \mid B \Rightarrow_{\mathcal{G}}^* 1\}.$$

Consequently, N_1 consists of those nonterminals which derive the empty word according to $\mathcal{G}$. Clearly, N_1 can be found by a procedure similar to those used in Theorem 3.1 to find T and R.

Now, let

$$A \longrightarrow x_1 \cdots x_n, \ x_i \in V,$$

be a production in $\mathcal{G}$. We replace it by the following set of productions

$$\{A \to \alpha \mid \alpha = \delta(x_1) \cdots \delta(x_n) \neq 1, \delta(x_i) \in \{x_i, 1\} \text{ if } x_i \in N_1 \text{ and } \delta(x_i) = x_i \text{ otherwise}\}.$$

Moreover, if $1 \in L(\mathcal{G})$, which can be tested, we add the production $S \to 1$. Let $\mathcal{G}_1$ be the CF grammar thus constructed. Clearly, the only possible erasing production in $\mathcal{G}_1$ is $S \to 1$.

Claim I. $L(\mathcal{G}_1) = L(\mathcal{G})$.

Proof. $L(\mathcal{G}_1) \subseteq L(\mathcal{G})$. Let $w \in L(\mathcal{G}_1)$. If $w = 1$, then clearly $1 \in L(\mathcal{G})$. Otherwise, by the construction of $\mathcal{G}_1$, we can translate a derivation

$$S \ \Rightarrow_{\mathcal{G}_1}^* \ w \qquad \text{to} \qquad S \ \Rightarrow_{\mathcal{G}}^* \ u_1 a_1 u_2 \cdots u_n a_n u_{n+1}$$

for some a_i´s and u_i´s such that $w = a_1 \cdots a_n$ and $u_i \in N_1^+$. Since, by the choice of N_1, $u_i \Rightarrow_{\mathcal{G}}^* 1$, we conclude that $w \in L(\mathcal{G})$.

$L(\mathcal{G}) \subseteq L(\mathcal{G}_1)$. Let $w \in L(\mathcal{G})$. Again the case $w = 1$ is clear. Otherwise we derive from a derivation

$$S \ \Rightarrow_{\mathcal{G}}^* \ w \in \Sigma^+ \tag{3.1}$$

a derivation

$$S \ \Rightarrow_{\mathcal{G}_1}^* \ w$$

as follows: in each step of (3.1) we erase all those occurrences of the right hand side of the considered production which contributes to w only the empty word. By the construction of $\mathcal{G}_1$, this yields a derivation according to $\mathcal{G}_1$.

So we have proved Claim I, and thus eliminated the erasing productions.

Next we introduce a new set of nonterminals

$$\{A_a \mid a \in \Sigma\},$$

add the productions

$$A_a \longrightarrow a, \quad a \in \Sigma$$

and change each occurrence of a terminal a by its counterpart A_a. Clearly, the grammar thus obtained is equivalent to the original one, and moreover, satisfies the requirements for the Chomsky normal form except that the productions of the form

$$A \longrightarrow B_1 \cdots B_n, \quad n \geq 1, \ B_i \in N \tag{3.2}$$

are still possible.

A production of the form (3.2) with $n \geq 3$ is easy to eliminate: Replace it by the following set of productions:

$$A \to B_1 C_1, \ C_1 \to B_2 C_2, \ldots, \ C_{n-1} \to B_{n-1} B_n,$$

where C_i´s are new nonterminals (distinct for each production of the form (3.2)). Obviously, this operation preserves the language.

Finally, we have to eliminate *chain* productions, i.e. productions of the form $A \to B$. At this point we assume that such productions are the only illegal productions in $\mathcal{G}$.

For each $A \in N$ we define (like N_1 above) the sets

$$N_A = \{B \in N \mid A \Rightarrow_{\mathcal{G}}^* B\},$$

and the set of productions

$$\mathcal{P}_A = \{A \to \alpha \mid \exists B \in N_A, \alpha \in \Sigma \cup N^2 : \ B \to \alpha\}.$$

Clearly, if $A \to \alpha \in \mathcal{P}_A$, then $A \Rightarrow_{\mathcal{G}}^* \alpha$, so that we can add $\mathcal{P}_A$ to $\mathcal{G}$ without changing the language it generates. Let $\mathcal{G}_+$ be obtained by adding all above $\mathcal{P}_A$´s to $\mathcal{G}$. Now, we omit all the chain productions from $\mathcal{G}_+$ to obtain a Chomsky normal form grammar $\mathcal{G}_{CNF}$.

Claim II. $L(\mathcal{G}_{CNF}) = L(\mathcal{G}_+) \quad (= L(\mathcal{G}))$.

Proof. $L(\mathcal{G}_{CNF}) \subseteq L(\mathcal{G}_+)$. Obvious since all productions of $\mathcal{G}_{CNF}$ are in $\mathcal{G}_+$.
$L(\mathcal{G}_+) \subseteq L(\mathcal{G}_{CNF})$. Let $w \in L(\mathcal{G}_+)$, i.e.

$$S \ \Rightarrow_{\mathcal{G}_+}^* w \in \Sigma^*. \tag{3.3}$$

If no chain productions are used in (3.3), then the derivation is according to $\mathcal{G}_{CNF}$, as well. If, in turn, (3.3) uses a chain production, say is of the form

$$S \ \Rightarrow_{\mathcal{G}_+}^* uAv \ \Rightarrow_{\mathcal{G}_+} uBv \ \Rightarrow_{\mathcal{G}_+}^* uCv \ \Rightarrow_{\mathcal{G}_+} u\alpha v \ \Rightarrow_{\mathcal{G}_+}^* w, \text{ with } \alpha \in \Sigma \cup N^2,$$

we can transform a derivation $uAv \Rightarrow_{\mathcal{G}_+}^* u\alpha v$ to a derivation $uAv \Rightarrow_{\mathcal{G}_{CNF}}^* u\alpha v$, and hence by induction $w \in L(\mathcal{G}_{CNF})$.

So we have proved Claim II and Theorem 3.2. □

By calling a language L *1-free* if $L \subseteq \Sigma^+$, we obtain from the proof of Claim I

Corollary 3.3. *Each 1-free CF language is generated by a nonerasing CF grammar i.e. by a CF grammar having no productions of the form* $A \to 1$.

We turn to prove that each CF language can be generated by a CF grammar in Greibach normal form, as well.

We need two auxiliary constructions.

Definition 3.5. Let $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ be a CF grammar. A production $A \to \beta$ is called an *A-production*. Further we call a production *left recursive*, if it is of the form

$$A \longrightarrow A\alpha, \quad \text{with } \alpha \in V^*.$$

In the following constructions we eliminate from $\mathcal{G}$ I) a nonterminating A-production II) all left recursive A-productions.

I. *Elimination of a nonterminating A-production.* Let $A \to \alpha_1 B \alpha_2$ be an A-production and $B \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_r$ be all B-productions of $\mathcal{G}$. If we replace the production $A \to \alpha_1 B \alpha_2$ by the productions $A \to \alpha_1\beta_1\alpha_2 \mid \alpha_1\beta_2\alpha_2 \mid \cdots \mid \alpha_1\beta_r\alpha_2$, then the language remains unchanged.

Proof. Let $\mathcal{G}_1$ be the grammar obtained as above from $\mathcal{G}$.

$L(\mathcal{G}_1) \subseteq L(\mathcal{G})$. A use of production $A \to \alpha_1\beta_i\alpha_2$ can be simulated by a derivation of $\mathcal{G}$ as follows: $A \Rightarrow_{\mathcal{G}} \alpha_1 B\alpha_2 \Rightarrow_{\mathcal{G}} \alpha_1\beta_i\alpha_2$.

$L(\mathcal{G}) \subseteq L(\mathcal{G}_1)$. If a production $a \to \alpha_1 B\alpha_2$ is used in a terminal derivation, then the above occurrence of B must be replaced later by some β_i. Consequently, the above derivation of $\mathcal{G}$ can be simulated by $\mathcal{G}_1$.

II. *Elimination of all left recursive A-productions.* Let $A \to A\alpha_1 \mid \cdots \mid A\alpha_r$ be the set of all left recursive A-productions of $\mathcal{G}$ and $A \to \beta_1 \mid \cdots \mid \beta_s$ the set of all other A-productions of $\mathcal{G}$. Let $\mathcal{G}_1 = (V \cup \{B\}, \Sigma, \mathcal{P}_1, S)$ be a CF grammar, where B is a new nonterminal, and $\mathcal{P}_1$ is obtained from $\mathcal{P}$ by replacing all A-productions by the productions

$$\begin{aligned} A &\longrightarrow \beta_i \mid \beta_i B, \quad i = 1, \ldots, s, \\ B &\longrightarrow \alpha_i \mid \alpha_i B, \quad i = 1, \ldots, r. \end{aligned}$$

Then $L(\mathcal{G}_1) = L(\mathcal{G})$.

Proof. $L(\mathcal{G}) \subseteq L(\mathcal{G}_1)$ follows from the facts that in a leftmost terminal derivation of $\mathcal{G}$ a sequence of productions $A \to A\alpha_i$ is followed by a production $A \to \beta_j$ and that the sequence

$$A \Rightarrow A\alpha_{i_1} \Rightarrow A\alpha_{i_2}\alpha_{i_1} \Rightarrow \ldots \Rightarrow A\alpha_{i_p}\cdots\alpha_{i_1} \Rightarrow \beta_j\alpha_{i_p}\cdots\alpha_{i_1}$$

can be replaced in $\mathcal{G}_1$ by

$$A \Rightarrow \beta_j B \Rightarrow \beta_j\alpha_{i_p}B \Rightarrow \ldots \Rightarrow \beta_j\alpha_{i_p}\cdots\alpha_{i_1}.$$

$L(\mathcal{G}_1) \subseteq L(\mathcal{G})$ follows since the above argument can be reversed.

Now, we are ready for

Theorem 3.4. *Each CF language can be generated by a CF grammar $\mathcal{G}_{GNF}$ in Greibach normal form. Moreover, such a $\mathcal{G}_{GNF}$ can be effectively found.*

Proof. Let L be generated by a CF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ in Chomsky normal form. If $1 \in L(\mathcal{G})$, then $\mathcal{G}$ contains a production $S \to 1$, and deleting it we get Chomsky normal form grammar for $L \setminus \{1\}$. If we can construct a Greibach normal form grammar for this language, we get it also for L by adding $S \to 1$ and possibly choosing a new start symbol. Therefore, we can assume that L is 1-free.

Let

$$N = \{A_1, A_2, \ldots, A_m\}.$$

In the *first step* we modify $\mathcal{G}$ such that

$$\text{if} \quad A_i \to A_j\gamma \quad \text{is a production, then} \quad j > i. \tag{3.4}$$

This is achieved by applying procedures I and II repeatedly starting from A_1 and proceeding to A_m:

For A_1 it is enough to apply II once to obtain a grammar satisfying (3.4). Assume that we already reached a situation, where (3.4) is satisfied for $i < k$.

If $A_k \to A_j\gamma$, with $j < k$, is a production in our grammar constructed at this point, we apply procedure I to eliminate it, or more precisely to replace it by new productions

which are obtained by substituting for A_j all possible right hand sides of A_j-productions. Since A_j´s, for $j < k$, already satisfy (3.4), after at most $k-1$ applications of procedure I (for all choices) we obtain a grammar, where productions $A_k \to A_j\gamma$, $j < k$, are replaced by productions of the form $A_k \to A_j\gamma$, with $j \geq k$ (and of the form $A_k \to a\gamma$, with $a \in \Sigma$). Now productions of the form $A_k \to A_k\gamma$ can be eliminated simultaneously by procedure II by introducing a new nonterminal B_k.

All in all this first step leads to a grammar, where productions are of the following forms:

(i) $A_i \longrightarrow A_j\gamma, \quad j > i,$

(ii) $A_i \longrightarrow a\gamma, \quad$ with $a \in \Sigma,$

(iii) $B_i \longrightarrow \gamma \in (V \cup \{B_1, \ldots, B_i\})^*.$

In the *second step* we change A_i-productions to Greibach normal form. We first note: γ´s in (i) and (ii) are in

$$(N \cup \{B_1, \ldots, B_m\})^*.$$

This follows, since for the original grammar this is true, since it was in CNF, and the procedures lead from productions in this form to new ones being still in the same form. Consequently, A_m-productions are in GNF, since there are none of the form (i). Moreover, A_{m-1}-productions can be replaced by productions in GNF by applying procedure I once. Repeating in the same way all A_i-productions can be replaced by productions in GNF.

In the *third step* we have to deal with new B_i-productions. B_i-productions are introduced in procedure II, when they are created from A_i-productions by the rule: $A_i \to A_i\alpha \;\Rightarrow\; B_i \to \alpha \mid \alpha B_i$. So we have to analyze right hand sides of A_i-productions. As we already noted they are in $(\{1\} \cup \Sigma)(N \cup \{B_1, \ldots, B_m\})^*$. Moreover, if such a production does not start with a terminal, it starts, by the same argument we used to conclude the form of γ above, with two A-nonterminals. Consequently, each B_i-production is either in a GNF or starts with some A_i, so that after one application of procedure I it can be replaced by GNF productions.

So we have completed our construction and found a Greibach normal form grammar for L. □

One can strengthen Theorem 3.4 to

Proposition 3.5. *Each CF language can be generated by a CF grammar in standard* 2*-form.*

Remark 3.2. The normal forms we proved are useful to prove properties of CF languages. For example, CNF guarantees that CF languages can be generated by very regular derivation trees: each node has either exactly two nonterminal descendants or only one terminal descendant.

GNF and standard 2-form resemble RL grammars, since productions are of the form

$$A \longrightarrow aB_1 \cdots B_n, \quad n \geq 0, a \in \Sigma, B_i \in N.$$

In particular, standard 2-form grammar can contain, besides RL productions $A \to a$ and $A \to aB$, only productions of the form $A \to aBC$. This, however, makes a big

difference in generating power, since, as we shall see, "the family of CF languages is much larger than the family of regular languages".

Note also that GNF shows that CF languages can be generated in *real time*, i.e. in the leftmost derivation of a terminal word each derivation step produces a terminal letter.

3.2 Properties of CF languages

In this section we consider properties of CF languages. Some further properties are shown later after showing that the family of CF languages can be characterized as languages accepted by a certain type of automata — pushdown automata. The properties considered are *closure properties*, in order to be able to show that languages are CF, *pumping properties*, in order to show that languages are not CF, and *decidability* properties.

We start with closure properties.

Definition 3.6. We say that a transduction $\delta : \Sigma^* \to \Delta^*$ is a *substitution*, if it satisfies $\delta(1) = \{1\}$ and

$$\delta(ww') = \delta(w) \cdot \delta(w') \quad \forall w, w' \in \Sigma^*.$$

Consequently, as in the case of a morphism, a substitution δ is completely defined by the languages $\delta(a)$, $a \in \Sigma$. In fact, δ can be viewed as a morphism from Σ^* into the monoid of all languages over Δ. A substitution δ is *finite*, *regular* or *context–free*, if languages $\delta(a)$, for $a \in \Sigma$ are so.

Theorem 3.6. *The family of CF languages is closed under CF substitutions.*

Proof. Let $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ be a CF grammar and $\delta : \Sigma^* \to \Delta^*$ a CF substitution, i.e. each $\delta(a)$, for $a \in \Sigma$, is CF, say generated by

$$\mathcal{G}_a = (V_a, \Delta, \mathcal{P}_a, S_a).$$

Assuming that (as we can)

$$(V_a \setminus \Delta) \cap V = \emptyset \quad \text{and} \quad (V_a \setminus \Delta) \cap (V_b \setminus \Delta) = \emptyset, \quad \forall a, b \in \Sigma,$$

we define a CF grammar

$$\mathcal{G}_\delta = (V \cup \bigcup_{a \in \Sigma} V_a, \Delta, \mathcal{P}_\delta \cup \bigcup_{a \in \Sigma} \mathcal{P}_a, S),$$

where $\mathcal{P}_\delta$ is obtained from $\mathcal{P}$ by replacing each occurrence of any terminal a in each production of $\mathcal{P}$ by S_a.

It follows immediately that $L(\mathcal{G}_\delta) = \delta\big(L(\mathcal{G})\big)$. □

Corollary 3.7. *A morphic image of a CF language is CF.*

Theorem 3.8. *The family of CF languages is closed under rational operations.*

Proof. Let $L_i = L(\mathcal{G}_i)$ for CF grammars $\mathcal{G}_i = (V_i, \Sigma, \mathcal{P}_i, S_i)$, $i = 1, 2$, where $(V_1 \setminus \Sigma_1) \cap (V_2 \setminus \Sigma_2) = \emptyset$. Then CF grammars for the languages $L_1 \cup L_2$, $L_1 L_2$ and L_1^* can be defined, respectively, as follows:

$$\mathcal{G}_\cup = (V_1 \cup V_2 \cup \{S\}, \Sigma, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \to S_1 \mid S_2\}, S),$$
$$\mathcal{G}_\cdot = (V_1 \cup V_2 \cup \{S\}, \Sigma, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \to S_1 S_2\}, S),$$

and

$$\mathcal{G}_* = (V_1 \cup \{S\}, \Sigma, \mathcal{P}_1 \cup \{S \to S S_1 \mid S_1 \mid 1\}, S),$$

where S is a new symbol.

It is obvious that the above constructions work as intended. □

Since for any language L we have $L^+ = LL^*$, we obtain

Corollary 3.9. *The family of CF languages is closed under 1-free iteration.*

The closure under the other Boolean operations than the union does not hold true for context–free languages, as we shall see as a consequence of pumping properties of these languages, which we start to consider now.

We recall that a pumping lemma for regular languages says that, if L is regular then for each long enough word z there exists a factorization $z = uxv$, with $x \neq 1$, such that $ux^*v \subseteq L$. Further as shown by the CF language $L = \{a^n b^n \mid n \geq 0\}$ this does not hold for CF languages. However, each word z of L can be factorized as $z = uxwyz$ such that $ux^n wy^n z \in L$ with $xy \neq 1$, for all $n \geq 0$. Indeed, take $u = w = z = 1$, $x = a$ and $y = b$. Consequently, in L one can pump simultaneously in two places! This is true for CF languages, in general, as we shall now show.

Actually, we prove a CF pumping lemma in a stronger form. In what follows, we are allowed to specify some positions, i.e. occurrences of letters, in a given word — these are called *marked*.

Theorem 3.10 (Iteration Theorem for CF languages). *Let L be a CF language. There exists a constant m such that for any word z in L if we mark at least m positions in z, we can factorize z as $z = uxwyv$ such that*

(i) x and y contain together at least one marked position,

(ii) xwy contains at most m marked positions, and

(iii) $ux^n wy^n v \in L$ for all $n \geq 0$.

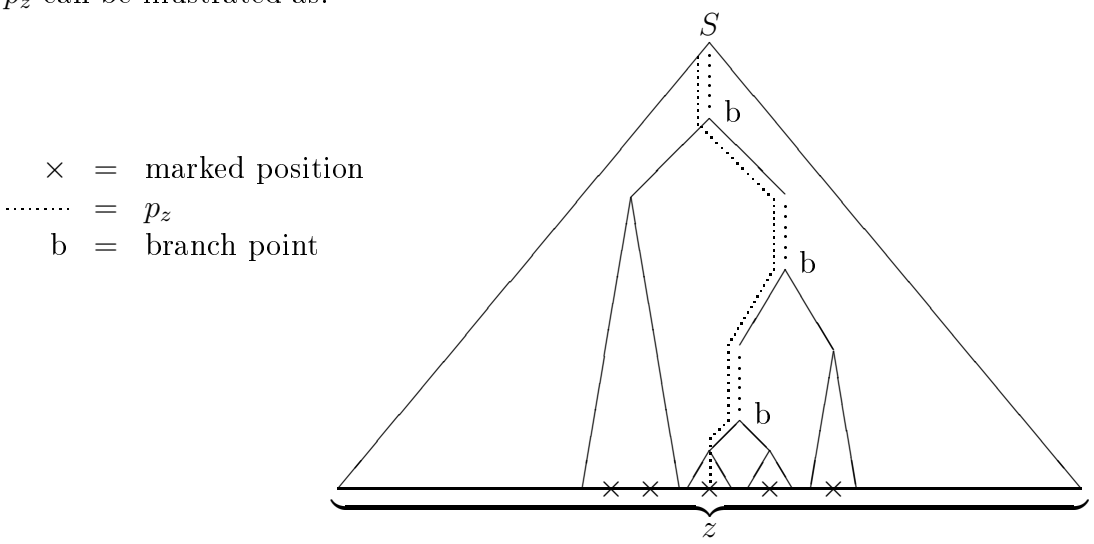
Proof. Assume that $L \setminus \{1\}$ is generated by Chomsky normal form grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. Let $k = |V \setminus \Sigma|$ and set $m = 2^{k+1}$.

Let $z \in L$ and assume that there exist at least m marked positions in z (hence $|z| \geq m$). We consider the derivation tree D_z of z. We construct a particular path p_z in D_z as follows:

- The root of D_z, that is S, is in p_z;
- If A is in p_z and it has two nonterminal descendants B and C, then that one which "contributes" more marked positions to z is in p_z, and in the case they contribute equally many we can choose either B or C arbitrarily;

– If A is in p_z and it has just one terminal descendant, then that is in p_z.

Clearly, since $\mathcal{G}$ is in CNF p_z is well defined. Further "what X contributes to z" is that factor of z, which is the yield of the subtree of D_z starting at X. Finally, we say that a nonterminal A above (as a node labeled by A in p_z) is a *branch point*, if it has two descendants and both of those contribute some marked positions to z. The construction of p_z can be illustrated as:



Now, a *crucial observation* is that each branch point contributes to z at least half as many marked positions as its previous branch point. By the choice of m there are at least 2^{k+1} marked positions in z all of which are descendants of S. So in p_z there are at least $k+1$ branch points. Consequently, among the $k+1$ last branch points there must be two with the same nonterminal label, say v_1 and v_2 with v_1 closer to the root and both labeled by A.

Let x, y and w be words in Σ^* such that w and xwy are yields of the subtrees of D_z starting at v_2 and v_1, respectively. Therefore

$$A \Rightarrow_{\mathcal{G}}^* w \quad \text{and} \quad A \Rightarrow_{\mathcal{G}}^* xAy.$$

Moreover, since v_1 is among the last $k+1$ branch points in p_z we conclude from our crucial observation that xwy contains at most 2^{k+1} marked positions. Hence, condition (ii) in Theorem is satisfied. On the other hand, v_1 is a branch point, so both of its descendants contribute at least one marked position. Therefore so does at least one of the words x or y, showing that condition (i) is satisfied, too. Finally, condition (iii) is fulfilled if we choose words u and v such that $uxwyv = z$. Indeed, then for any $n \geq 0$ we have

$$S \Rightarrow_{\mathcal{G}}^* uAv \Rightarrow_{\mathcal{G}}^* ux^nAy^nv \Rightarrow_{\mathcal{G}}^* ux^nwy^nv. \qquad \square$$

Often the above theorem is proved by assuming that all positions in z are marked — then instead of marked positions we can consider simply lengths of words:

Corollary 3.11 (Pumping Lemma for CF languages). *For each CF language L there exists a constant m such that any word z of the length at least m has a factorization $z = uxwyv$ such that*

(i) $|xy| \geq 1$,

(ii) $|xwy| \leq m$, *and*

(iii) $ux^n wy^n v \in L$ *for all* $n \geq 0$.

Remark 3.3. The use of a CNF grammar in the above proof is not necessary, but it makes the proof neater.

By Pumping Lemma we can prove that some languages are not CF.

Example 3.4. We show that the language

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

is not CF. Marking b´s we could slightly decrease the number of different cases needed to be analyzed in the below proof. However, Iteration Theorem would not make the proof essentially simpler than the ordinary Pumping Lemma.

The ordinary Pumping Lemma works here: For some large enough n we can write

$$a^n b^n c^n = uxwyv \quad \text{with } |xy| \geq 1$$

and

$$ux^m wy^m v \in L \quad \text{for all } m \geq 0.$$

We have to analyze the following cases:

1) If x or y is in d^+e^+, with $d, e \in \{a, b, c\}$, $d \neq e$, then clearly $ux^2wy^2v \notin a^*b^*c^*$, a contradiction.

2) If $x, y \in a^* \cup b^* \cup c^*$, then either $|uwv|_a \neq |uwv|_b$ or $|uwv|_a \neq |uwv|_c$, a contradiction.

Remark 3.4. There are languages which can be shown non–CF by Iteration Theorem, but not by Pumping Lemma. An example of such a language is $\{a^*bc\} \cup \{a^p b a^n c a^n \mid p \in \mathbb{P}, n \geq 0\}$, cf. Exercises.

Now we are ready for some nonclosure properties.

Theorem 3.12. *The family of CF languages is not closed under intersection or complementation.*

Proof. Let

$$L_1 = \{a^i b^i c^j \mid i, j \geq 1\}$$

and

$$L_2 = \{a^i b^j c^j \mid i, j \geq 1\}.$$

Then

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\},$$

which by Example 3.4 is non–CF (of course, it does not matter whether the index goes from 0 or 1).

However, L_1 is generated by CF productions

$$S \longrightarrow AC, \quad A \longrightarrow aAb \mid 1, \quad C \longrightarrow cC \mid 1.$$

Similarly, L_2 is CF. So we have established the nonclosure under intersection.

The nonclosure under complementation follows from above and Theorem 3.8:

$$L \cap K = \Sigma^* \setminus \big((\Sigma^* \setminus L) \cup (\Sigma^* \setminus K)\big).$$

□

Now we use pumping properties of CF languages to establish a connection between regular and context–free languages — Parikh Theorem.

Definition 3.7. We call a mapping

$$\pi : \Sigma^* \to \mathbb{N}^{|\Sigma|}, \quad \pi(w) = (|w|_{a_1}, \ldots, |w|_{a_n}),$$

where $\Sigma = \{a_1, \ldots, a_n\}$, *Parikh mapping*. Further we call two languages L and L' *letter equivalent*, if their Parikh images coincide, i.e. $\pi(L) = \pi(L')$.

Parikh Theorem states that each CF language is letter equivalent to a regular language. We need as an auxiliary result the following modification of pumping lemma.

Lemma 3.13. *Let $\mathcal{G}$ be a CF grammar in CNF and k a natural number. There exists a constant p (depending on $\mathcal{G}$ and k) such that for any word $z \in L(\mathcal{G})$ of the length at least p, there exists a derivation*

$$\begin{aligned} S \Rightarrow^* uAv \Rightarrow^* ux_1Ay_1v \Rightarrow^* ux_1x_2Ay_2y_1v \Rightarrow^* \ldots \Rightarrow^* ux_1\cdots x_kAy_k\cdots y_1v \\ \Rightarrow^* ux_1\cdots x_kwy_k\cdots y_1v = z \end{aligned} \tag{3.5}$$

for some $A \in N$ and words $x_i, y_i, u, v, w \in \Sigma^*$ such that

(i) $|x_iy_i| \geq 1$, *for* $i = 1, \ldots, k$,

(ii) $|x_1\cdots x_kwy_k\cdots y_1| \leq p$.

Proof. We first note that since $\mathcal{G}$ is in CNF, that is productions are either length increasing $A \to BC$ or terminating $A \to a$, there exists a constant q such that for any word z of length at least q any derivation tree contains a path repeating a nonterminal at least $k+1$ times, that is z has a derivation of the form (3.5), where condition (i) is satisfied since $\mathcal{G}$ is in CNF.

In the above derivation tree considered we can choose the path yielding (3.5) in such a way that the subtree starting at the topmost occurrence of A does not contain on any path any nonterminal more than k times (except that A occurs $k+1$ times as chosen). Of course, such a derivation (3.5) can be found. Then the length of $x_1\cdots x_kwy_k\cdots y_1$ would be bounded by a constant r $(= 2^{k\cdot|N|+1})$.

We can choose $p = \max(r, q)$. □

We still need one more notion. For a CF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ and a subset $Q \subseteq N = V \setminus \Sigma$, by a *derivation in Q* we mean a derivation of $\mathcal{G}$ which uses *all nonterminals of Q*, and *only nonterminals of Q*. We set

$$L(\mathcal{G}; Q) = \{w \in \Sigma^* \mid w \text{ has a derivation in } Q \text{ from } S\}.$$

Clearly,

$$L = \bigcup_{Q \subseteq N} L(\mathcal{G}; Q). \tag{3.6}$$

Theorem 3.14 (Parikh Theorem). *Each CF language is letter equivalent to a regular language.*

Proof. Let $L = L(\mathcal{G})$ for a CNF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. Further let $Q \subseteq N(= V \setminus \Sigma)$. Since the family of regular languages is closed under union, it follows from (3.6) that it is enough to show that $L(\mathcal{G}; Q)$ is letter equivalent to a regular language.

Let $k = |Q|$ and p the constant of Lemma 3.13 associated to $\mathcal{G}$ and k. We set

$$F = \{w \in \Sigma^* \mid |w| < p, w \in L(\mathcal{G}; Q)\}$$

and

$$T = \{w \in \Sigma^* \mid w = xy, A \Rightarrow_{\mathcal{G}}^* xAy \text{ using only nonterminals from Q}, A \in Q, 1 \leq |xy| < p\}.$$

Then F and T are finite, so that FT^* is regular. Hence it is enough to prove

Claim. $\pi(FT^*) = \pi\big(L(\mathcal{G}; Q)\big)$.

Proof of Claim. $\pi(FT^*) \subseteq \pi\big(L(\mathcal{G}; Q)\big)$: We have to show that for $z \in FT^*$, there exists $z' \in L(\mathcal{G}; Q)$ such that $\pi(z) = \pi(z')$. This is done by induction on i in $FT^* = \bigcup_{i \geq 0} FT^i$.

$i = 0$: Then $z \in F$ and so also $z \in L(\mathcal{G}; Q)$.

Induction step: Let $z \in FT^i$. We write $z = ft$ with $f \in FT^{i-1}$ and $t \in T$. We find, by induction hypothesis, an f' in $L(\mathcal{G}; Q)$ such that $\pi(f) = \pi(f')$. Now, by the choice of T, we have

$$t = xy \quad \text{and} \quad A \Rightarrow^* xAy \quad \text{for some } A \in Q.$$

But $f' \in L(\mathcal{G}; Q)$ and $A \in Q$, so that we have the following derivation in Q

$$S \Rightarrow^* uAv \Rightarrow^* uwv = f'.$$

Consequently, $uxwyv$ has a derivation in Q, and since clearly $\pi(uxwyv) = \pi(f't) = \pi(ft) = \pi(z)$ we are done.

$\pi\big(L(\mathcal{G}; Q)\big) \subseteq \pi(FT^*)$: Let $z \in L(\mathcal{G}; Q)$. Again we find, by induction on $|z|$, $z' \in FT^*$ such that $\pi(z) = \pi(z')$.

$|z| < p$: Then $z \in F$ and we are done.

Induction step: Assume that $|z| = n \geq p$. Now z has a derivation in Q: $S \Rightarrow^* z$ using exactly nonterminals from Q. By Lemma 3.13, we can write this in the form

$$\begin{aligned} S \Rightarrow^* uAv \Rightarrow^+ ux_1Ay_1v \Rightarrow^+ \ldots \Rightarrow^+ ux_1 \cdots x_kAy_k \cdots y_1v \\ \Rightarrow^+ ux_1 \cdots x_kwy_k \cdots y_1v = z, \end{aligned} \tag{3.7}$$

where, for each i, $1 \leq |x_iy_i| \leq p$. (Note that here we have used Lemma 3.13 not for the word z, but for its derivation $S \Rightarrow^* z$, which is by the proof of Lemma 3.13 OK).

Now, (3.7) is a derivation in Q. So all k nonterminals appear in (3.7). For each of the $k-1$ nonterminals in $Q \setminus \{A\}$ we pick one of the $k+2$ steps of (3.7) containing this nonterminal. There remains at least one step of the form

$$ux_1 \cdots x_iAy_i \cdots y_1v \Rightarrow^+ ux_1 \cdots x_{i+1}Ay_{i+1} \cdots y_1v,$$

which is not picked. By removing this we obtain a derivation in Q for the word $\bar{z} = ux_1 \cdots x_ix_{i+2} \cdots x_kwy_k \cdots y_{i+2}y_i \cdots y_1v \in \Sigma^*$, with $|\bar{z}| < |z|$. By induction hypothesis, there exists a $\bar{z}'$ in FT^* such that $\pi(\bar{z}) = \pi(\bar{z}')$. Set $z' = \bar{z}'x_iy_i$ and we are done: $z' \in FT^*$ and $\pi(z) = \pi(z')$. □

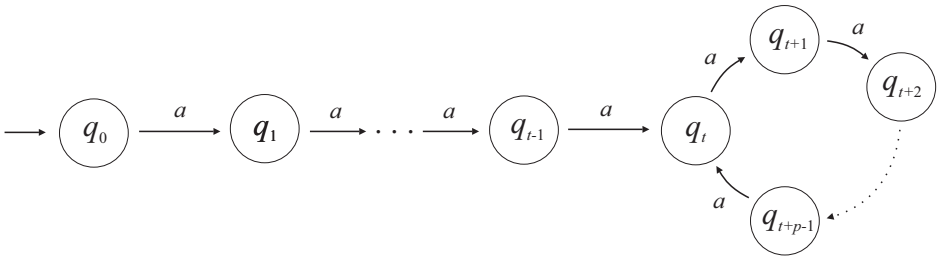
Since for words $w, w' \in a^*$ we have: $\pi(w) = \pi(w')$ iff $w = w'$, we obtain

Corollary 3.15. *Each CF language $L \subseteq a^*$ is regular.*

Example 3.5. We claim that unary CF language, i.e. CF languages over $\{a\}$, are *ultimately periodic*, i.e. of the form

$$\{a^{n_1}, \ldots, a^{n_i}\} \cup \{a^{n_{i+1}}, \ldots, a^{n_{i+j}}\}\{a^p\}^* \quad \text{for some } i, j \geq 0, \tag{3.8}$$

where $0 \leq n_1 < n_2 < \cdots < n_{i+j}$ and $p \geq 0$. By previous corollary, it suffices to prove this for regular languages. Let $L \subseteq a^*$ be accepted by a complete DFA $\mathcal{A}$. Then $\mathcal{A}$ is of the form:



(3.9)

Now, any choice of the final states gives a presentation of the form (3.8), and also conversely any presentation of the form (3.8) yields a DFA of the form (3.9) accepting the language.

Now, we turn to the decidability questions for CF languages. We consider the same questions we did for regular languages.

Theorem 3.16. *The membership, emptiness and finiteness problems are decidable for CF languages (given by CF grammars).*

Proof. Membership. Here a grammar $\mathcal{G}$ and a word $w \in \Sigma^*$ are given, and the algorithm has to decide "$w \in L(G)$?".

An algorithm: First change $\mathcal{G}$ to an equivalent GNF grammar $\mathcal{G}'$, by Theorem 3.4. Then decide whether $S \to 1$ is in $\mathcal{G}'$, if $w = 1$, or whether $\mathcal{G}'$ derives w in any derivation of the length $|w|$, if $w \neq 1$.

This algorithm clearly works correctly, but is very unefficient. Indeed, the transformation $\mathcal{G} \to \mathcal{G}'$ is not easy, and even worse there may exist $k^{|w|}$ leftmost derivations for words of length $|w|$, where k is the number of productions in $\mathcal{G}'$.

Emptiness. Clearly $L(\mathcal{G}) \neq \emptyset$ iff $\exists w \in \Sigma^* : S \Rightarrow_{\mathcal{G}}^* w$ iff S is terminating. The last property can be checked by a construction in the proof of Theorem 3.1.

Finiteness (or infiniteness). *An algorithm*:

1) Find for $\mathcal{G}$ an equivalent CNF grammar $\mathcal{G}'$;

2) Find for $\mathcal{G}'$ an equivalent reduced grammar $\mathcal{G}''$;

3) Check whether $\mathcal{G}''$ contains a nonterminal A satisfying

$$A \Rightarrow_{\mathcal{G}''}^* uAv \quad \text{for } u, v \in V^* \text{ and } uv \neq 1, \tag{3.10}$$

and if "yes", output "$L(\mathcal{G})$ is not finite", and otherwise "$L(\mathcal{G})$ is finite".

Parts 1) and 2) can be done by Theorems 3.2 and 3.1, respectively. Condition (3.10), in turn, can be tested by a construction in the proof of Theorem 3.1.

The correctness of the algorithm is seen as follows: If there exists an A satisfying (3.10), then $L(\mathcal{G})$ is infinite, since $\mathcal{G}''$ is reduced uv cannot be erased (since $\mathcal{G}''$ is in CNF), and we can pump (3.10) arbitrarily many times. On the other hand, if no A satisfying (3.10) exists, then, since $\mathcal{G}$ is in CNF, the paths in derivation trees are shorter than $|N|$. Hence, $L(\mathcal{G})$ is finite. □

Remark 3.5. Contrary to Theorem 3.16 many other problems for CF languages are algorithmically undecidable. Examples of such problems are:

Equivalence problem : $L(\mathcal{G}) \stackrel{?}{=} L(\mathcal{G}')$.

Ambiguity problem : Is a given CF grammar ambiguous?

Universality problem : $L(\mathcal{G}) \stackrel{?}{=} \Sigma^*$.

Later we shall have tools to show such results.

We conclude this section with a more practical solution for the membership problem.

Theorem 3.17 (Cocke–Younger–Kasami -algorithm). *The membership problem for CF languages can be solved in cubic time, i.e. in* $\mathcal{O}(|w|^3)$.

Proof. First we transform a grammar generating L into a CNF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. (This requires only a constant time in terms of $|w|$!)

We are given $w = a_1 \cdots a_n$, with $a_i \in \Sigma$. We denote by α_{ij} the factor of w starting at the position i and of length j. Therefore $\alpha_{ij} = a_i \cdots a_{i+j-1}$. So if we let $j = 1, \ldots, n$, then $i = 1, \ldots n - j + 1$. The basic idea is to find all nonterminals A satisfying

$$A \Rightarrow_{\mathcal{G}}^* \alpha_{ij}. \tag{3.11}$$

This is done inductively on j and keeping a list of sets of nonterminals computed earlier. In more details, we build the upper left half of the matrix, where entry (i, j) consists of exactly those A´s satisfying (3.11). Hence, the question "$w \in L(\mathcal{G})$?" can be answered by checking whether S is in the entry $(1, n)$.

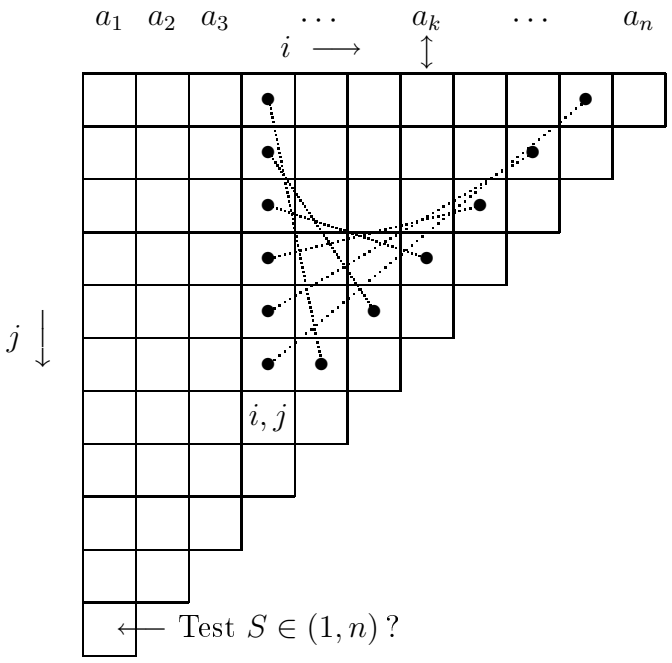
The construction of the matrix is illustrated below:

The first row is easy to fill. Since $\mathcal{G}$ is in CNF, A is in $(i, 1)$ iff $A \to a_i \in \mathcal{P}$. Now, assuming that the $j - 1$ first rows are filled, the next one is filled as follows:

A is in (i, j) iff
$A \Rightarrow^* \alpha_{ij}$ iff (since $\mathcal{G}$ is in CNF)
$A \longrightarrow BC \in \mathcal{P}$ and $\exists k \in \{1, \ldots, j-1\}$: $B \Rightarrow^* \alpha_{ik}$ and $C \Rightarrow^* \alpha_{i+k,j-k}$ iff
$A \longrightarrow BC \in \mathcal{P}$ and $\exists k \in \{1, \ldots, j-1\}$: B in (i, k) and C in $(i + k, j - k)$.

The correctness of the algorithm follows directly from the construction. The complexity of the algorithm is estimated as follows:

- We have to compute the value of $\mathcal{O}(n^2)$ entries;
- Each value can be computed by comparing at most n pairs of earlier computed entries;



Lines ●·····● show the entries needed to be checked in order to fill the entry (i, j).

Arrow $\longleftarrow$ shows how entries $(i, 1)$ are filled.

– Comparing two entries in the above can be done in a constant time on n.

Hence, the complexity is $\mathcal{O}(n^3)$. □

Remark 3.6. There are methods to improve the above algorithm, but no algorithm with complexity $\mathcal{O}(n^2)$ is known for the problem.

Remark 3.7. The above algorithm can be used to find a derivation tree for w also in time $\mathcal{O}(n^3)$.

Remark 3.8. The method used to construct the above algorithm is that of the *dynamical programming*: a problem is solved by solving smaller instances and keeping a list of their solutions.

3.3 Pushdown automata

In this section we define a family of automata — pushdown automata — which characterizes the family of CF languages. These automata are generalizations of FA such that in addition to a finite memory in states, they have an additional, potentially infinite, memory, which however is of a very special type, so called *LIFO–type* (last–in–first–out). Moreover, these devices are nondeterministic and capable of reading empty word. Here the deterministic variant (dpda) is strictly less powerful than the general model (pda).

Definition 3.8. *Pushdown automaton* $\mathcal{M}$, pda for short, is a seventuple

$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F),$$

where

(i) Q is a finite set of *states*,

(ii) Σ is a finite *input alphabet*,

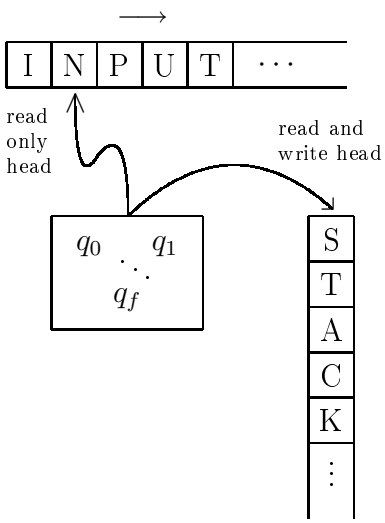
(iii) Γ is a finite *stack alphabet*,

(iv) δ is a *transition* function $Q\times(\Sigma\cup\{1\})\times\Gamma \to 2^{Q\times\Gamma^*}$,

(v) $q_0 \in Q$ is the *initial state*,

(vi) $z_0 \in \Gamma$ is the initial *stack symbol*,

(vii) $F \subseteq Q$ is the set of *final states*.



As in the case of an FA, a pda can be illustrated as shown in the figure above. Transitions of $\mathcal{M}$ are of the form

$$(p, \alpha, z, q, \gamma) \in Q \times (\Sigma \cup \{1\}) \times \Gamma \times Q \times \Gamma^* \tag{3.12}$$

and are interpreted as follows: "When in state p, reading α and z is the topmost symbol in the stack, then $\mathcal{M}$ can move to state q, replace z by γ, and stay in the same square of the input tape or move one step to the right depending on whether $\alpha = 1$ or $\alpha \in \Sigma$".

For a triple (p, α, z) there may be several pairs (q, γ) such that $(p, \alpha, z, q, \gamma)$ is a transition — hence $\mathcal{M}$ is *nondeterministic*. Also the case $\alpha = 1$ is allowed — hence $\mathcal{M}$ may have *1-transitions*.

A configuration of $\mathcal{M}$ during a computation, its *instantaneous description*, ID for short, is defined as a triple $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$, corresponding to a current state, so far unread part of the input and the contents of the stack.

Definition 3.9. On the set of ID´s we define a relation $\vdash_{\mathcal{M}}$ as follows:

$$(p, \alpha w, z\beta) \ \vdash_{\mathcal{M}} \ (q, w, \gamma\beta) \qquad \text{if} \qquad (q, \gamma) \in \delta(p, \alpha, z).$$

By $\vdash^*_{\mathcal{M}}$ we mean the reflexive and transitive closure of $\vdash_{\mathcal{M}}$. If ID $\vdash_{\mathcal{M}}$ ID', we say that $\mathcal{M}$ *derives directly* ID' from ID, corresponding a *one step computation*. Consequently, $\vdash^i_{\mathcal{M}}$ means an i step computation according to $\mathcal{M}$. The *initial* configuration of $\mathcal{M}$ is (q_0, w, z_0).

Definition 3.10. A language can be associated with a pda $\mathcal{M}$ in a number of different ways. A *language accepted by* $\mathcal{M}$ *(with final states)* is

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid (q_0, w, z_0) \vdash^*_{\mathcal{M}} (q, 1, \gamma) \text{ with } q \in F, \gamma \in \Gamma^*\}.$$

A *language accepted by* $\mathcal{M}$ *with empty stack* (resp. *with empty stack and final states*) is

$$\begin{aligned} N(\mathcal{M}) &= \{w \in \Sigma^* \mid (q_0, w, z_0) \vdash^*_{\mathcal{M}} (q, 1, 1) \text{ with } q \in Q\} \\ (\text{resp. } T(\mathcal{M}) &= \{w \in \Sigma^* \mid (q_0, w, z_0) \vdash^*_{\mathcal{M}} (q, 1, 1) \text{ with } q \in F\}\ ). \end{aligned}$$

Remark 3.9. All three models of the acceptance define exactly the same family of languages, namely the family of CF languages, as we shall see. If the acceptance is not specified in details, we mean the acceptance with final states. This makes pda´s natural extensions of FA´s. Of course, the acceptance with the empty stack and final states would be mathematically nicest — however, it is not so suitable when showing that particular languages are CF.

Example 3.6. Let us search for a pda for the language $\{a^n b^n \mid n \geq 0\} = L$. The idea is clear: pda counts the number of a´s in the stack, so that it can accept when the number of b´s equal to that of a´s. More formally, let $\mathcal{M} = (\{q_0, q_a, q_b, q_f\}, \{a, b\}, \{z_0, a, b\}, \delta, q_0, z_0, \{q_f\})$ with the transitions:

$$\begin{aligned}
(q_0, 1, z_0) &\longrightarrow (q_f, 1) \quad \text{(accepts 1)} \\
(q_0, a, z_0) &\longrightarrow (q_a, az_0) \\
(q_a, a, a) &\longrightarrow (q_a, aa) \\
(q_a, b, a) &\longrightarrow (q_b, 1) \\
(q_b, b, a) &\longrightarrow (q_b, 1) \\
(q_b, 1, z_0) &\longrightarrow (q_f, 1).
\end{aligned}$$

Clearly, $L(\mathcal{M}) = N(\mathcal{M}) = T(\mathcal{M}) = L$.

Example 3.7. The language $\{wcw^R \mid w \in \{a, b\}^+\}$ is accepted by a pda $\mathcal{M}$ having transitions:

$$\begin{aligned}
(q_0, x, z_0) &\longrightarrow (q_0, xz_0) \quad \text{for } x \in \{a, b\}, \\
(q_0, x, y) &\longrightarrow (q_0, xy) \quad \text{for } x, y \in \{a, b\}, \\
(q_0, c, x) &\longrightarrow (q_1, x) \quad \text{for } x \in \{a, b\}, \\
(q_1, x, x) &\longrightarrow (q_1, 1) \quad \text{for } x \in \{a, b\}, \\
(q_1, 1, z_0) &\longrightarrow (q_f, 1),
\end{aligned}$$

with q_0 and q_f the initial and final state, respectively. Again $L(\mathcal{M}) = N(\mathcal{M}) = T(\mathcal{M})$. That $\mathcal{M}$ works correctly is clear: First when reading w it is *pushed* to the stack, and when detecting c in the input $\mathcal{M}$ starts to *pop* symbols from the stack, and at the same time compares that the rest of the input word is exactly the reverse of the word pushed to the stack.

If we change the third transition (schema) by

$$(q_0, x, x) \longrightarrow (q_1, 1) \quad \text{for } x \in \{a, b\},$$

we obtain pda $\mathcal{M}'$ accepting the language $L' = \{ww^R \mid w \in \{a, b\}^*\}$. The argument is the same. Only in $\mathcal{M}$ we *detect* the middle of the input, while in $\mathcal{M}'$ we *guess* it, and then confirm that the guess was correct. $\mathcal{M}'$ is clearly nondeterministic, while $\mathcal{M}$ is deterministic in the sense we define precisely later.

Next we show that all three different models of the acceptance lead to the same family of languages.

Lemma 3.18. *For each pda $\mathcal{M}$ there exists a pda $\mathcal{M}'$ such that $L(\mathcal{M}) = N(\mathcal{M}')$.*

Proof. Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. We have to construct a pda $\mathcal{M}'$ such that

- $\mathcal{M}'$ simulates $\mathcal{M}$, and
- whenever $\mathcal{M}$ enters to a final state, then $\mathcal{M}'$ empties its stack (and otherwise the stack of $\mathcal{M}'$ is nonempty).

We set $\mathcal{M}' = (Q \cup \{q_e, q_0'\}, \Sigma, \Gamma \cup \{x_0\}, \delta', q_0', x_0, -)$ with the transitions:

(i) $\delta'(q_0', 1, x_0) = (q_0, z_0 x_0)$,

(ii) $\delta'(q, a, z) = \delta(q, a, z)$,

(iii) $\delta'(q, 1, z) = (q_e, 1)$ for $q \in F, z \in \Gamma \cup \{x_0\}$,

(iv) $\delta'(q_e, 1, z) = (q_e, 1)$ $z \in \Gamma \cup \{x_0\}$.

Note that the set of final states of $\mathcal{M}'$ need not be defined.

Claim. $L(\mathcal{M}) = N(\mathcal{M}')$.

Proof of Claim: $L(\mathcal{M}) \subseteq N(\mathcal{M}')$. Let $w \in L(\mathcal{M})$, i.e.

$$(q_0, w, z_0) \vdash_{\mathcal{M}}^* (q, 1, \gamma) \quad \text{for some } q \in F \text{ and } \gamma \in \Gamma^*.$$

Then according to $\mathcal{M}'$ we have:

$$(q_0', w, x_0) \vdash_{\mathcal{M}'} (q_0, w, z_0 x_0) \vdash_{\mathcal{M}'}^* (q, 1, \gamma x_0) \vdash_{\mathcal{M}'} (q_e, 1, \gamma') \vdash_{\mathcal{M}'}^* (q_e, 1, 1),$$

where $\gamma' = 1$ if $\gamma = 1$, and otherwise $\gamma' = c^{-1}\gamma x_0$, where c is the first symbol of γ.

It follows that $w \in N(\mathcal{M}')$.

$N(\mathcal{M}') \subseteq L(\mathcal{M})$. Let $w \in N(\mathcal{M}')$, i.e. $(q_0', w, x_0) \vdash_{\mathcal{M}'}^* (q, 1, 1)$ for some q. The above computation starts with a rule (i) which leaves the symbol x_0 to the bottom of the stack. Since the computation is accepting it has to be removed, and this can be done only by rules of (iii) or (iv), that is in the state q_e. In order to enter to state q_e, $\mathcal{M}'$ has to reach a final state of $\mathcal{M}$. None of the rules (i), (iii) or (iv) consumes any input symbol. So the computation from q_0 to a final state of $\mathcal{M}$ according to $\mathcal{M}'$ is an accepting computation of w in $\mathcal{M}$. Hence $w \in L(\mathcal{M})$. □

Lemma 3.19. *For each pda $\mathcal{M}$ there exists a pda $\mathcal{M}'$ such that $N(\mathcal{M}) = L(\mathcal{M}')$.*

Proof. Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, -)$ be a pda. Now we have to construct a pda $\mathcal{M}'$ such that

- $\mathcal{M}'$ simulates $\mathcal{M}$, and
- whenever $\mathcal{M}$ empties the stack (and stops), then $\mathcal{M}'$ moves to its final state (and this is the only way to go to a final state).

We define $\mathcal{M}' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{x_0\}, \delta', x_0, \{q_f\})$, where δ' is as follows:

(i) $\delta'(q_0', 1, x_0) = (q_0, z_0 x_0)$,

(ii) $\delta'(q, a, z) = \delta(q, a, z)$,

(iii) $\delta'(q, 1, x_0) = (q_f, 1)$.

Rule (i) leads from the initial ID of $\mathcal{M}'$ to that of $\mathcal{M}$, with the additional property that at the bottom of the stack is x_0. Then rule (ii) allows $\mathcal{M}'$ to simulate $\mathcal{M}$, and if $\mathcal{M}$ makes the stack empty, then and only then, (iii) becomes applicable allowing $\mathcal{M}'$ to enter to its final state without reading any symbols.

The above clearly shows that $N(\mathcal{M}) \subseteq L(\mathcal{M}')$. But since the above was the only way to reach a final state also $L(\mathcal{M}') \subseteq N(\mathcal{M})$. □

Theorem 3.20. *The three families of languages accepted by pda´s with*

(i) final states;

(ii) empty stack; or

(iii) final states and empty stack,

coincide.

Proof. The equivalence of (i) and (ii) follows from Lemmas 3.18 and 3.19. The equivalence of (ii) and (iii) is easy and left as an exercise. □

Next we prove the main result of this section.

Theorem 3.21. *A language $L \subseteq \Sigma^*$ is CF iff it is accepted by a pda.*

Proof. By Theorem 3.20 we may use the acceptance with the empty stack.

$\Rightarrow$: Let $L \subseteq \Sigma^*$ be CF. It is enough to prove the implication in the case $1 \notin L$, since we can easily construct — by introducing a new initial state — from a pda for $L \setminus \{1\}$ a pda for L.

We assume that L is generated by GNF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$. We define a pda $\mathcal{M} = (\{q\}, \Sigma, N, \delta, q, S, \emptyset)$, where

$$(q, \gamma) \in \delta(q, a, A) \qquad \text{iff} \qquad A \longrightarrow a\gamma \in \mathcal{P}.$$

The basic idea is that $\mathcal{M}$ simulates leftmost derivations of $\mathcal{G}$ by remembering the sequences of nonterminals in sentential forms of the derivations of $\mathcal{G}$ in its stack. More formally, we claim, that

$$S \ \Rightarrow_{\mathcal{G}}^* \ x\gamma \quad \text{with } x \in \Sigma^*, \gamma \in N^* \tag{3.13}$$

is a leftmost derivation of $\mathcal{G}$ iff

$$(q, x, S) \ \vdash_{\mathcal{M}}^* \ (q, 1, \gamma).$$

Assume first that $(q, x, S) \vdash_{\mathcal{M}}^i (q, 1, \gamma)$. We prove by induction on i, that $S \Rightarrow_{\mathcal{G}}^* x\gamma$. The case $i = 0$ is clear: $x = 1$, $\gamma = S$. For the induction step we write $x = ya$ and

$$(q, x, S) \ \vdash^{i-1} \ (q, a, \beta) \ \vdash \ (q, 1, \gamma). \tag{3.14}$$

Now $(q, y, S) \vdash^{i-1} (q, 1, \beta)$ so that, by the induction hypothesis, in $\mathcal{G}$ $S \Rightarrow^* y\beta$. By (3.14) and the construction of $\mathcal{M}$, there must be A in N such that $\beta = A\beta'$, $A \to a\eta$ is in $\mathcal{P}$ and $\gamma = \eta\beta'$. Consequently,

$$S \ \Rightarrow_{\mathcal{G}}^* \ yA\beta' \ \Rightarrow_{\mathcal{G}} \ ya\eta\beta' = x\gamma,$$

as required.

Secondly assume that $S \Rightarrow_{\mathcal{G}}^i x\gamma$ is a leftmost derivation with $x \in \Sigma^*$ and $\gamma \in N^*$. We prove, by induction on i, that $(q, x, S) \vdash_{\mathcal{M}}^* (q, \gamma)$. Again the case $i = 0$ is trivial. For the induction step we write $x = ya$ and

$$S \ \Rightarrow_{\mathcal{G}}^{i-1} \ yA\beta' \ \Rightarrow_{\mathcal{G}} \ ya\eta\beta' = x\gamma \quad \text{with } a \in \Sigma.$$

By the induction hypothesis

$$(q, y, S) \vdash^*_{\mathcal{M}} (q, 1, A\beta'),$$

and so also

$$(q, ya, S) \vdash^*_{\mathcal{M}} (q, a, A\beta').$$

Now, since $A \to a\eta \in \mathcal{P}$ we obtain from the construction of $\mathcal{M}$ that

$$(q, x, S) \vdash^*_{\mathcal{M}} (q, a, A\beta') \vdash_{\mathcal{M}} (q, 1, \eta\beta') = (q, 1, \gamma)$$

as required.

To conclude, we note that (3.13) implies, when choosing $\gamma = 1$, that $L(\mathcal{G}) = N(\mathcal{M})$, so that L is accepted by a pda with the empty stack.

$\Leftarrow$: Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \emptyset)$ be a pda such that $L = N(\mathcal{M})$. We construct a CF grammar $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$, where

$$V = \{S\} \cup (Q \times \Gamma \times Q) \cup \Sigma,$$

and $\mathcal{P}$ consists of productions:

(i) $S \longrightarrow [q_0, z_0, q]$ for $q \in Q$,

(ii) $[q, A, q_{m+1}] \longrightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_m, B_m, q_{m+1}]$ for $q, q_1, \ldots, q_{m+1} \in Q$, $a \in \Sigma \cup \{1\}$ and $A, B_1, \ldots, B_m \in \Gamma$ such that $(q_1, B_1 \cdots B_m) \in \delta(q, a, A)$.

In (ii), if $m = 0$, then the production is of the form $[q, A, q_1] \to a$.

The idea of the above construction is as follows. A nonterminal $[p, A, q]$ in $\mathcal{G}$ is eliminated in a leftmost derivation, and at the same time a terminal word x is produced iff x causes in $\mathcal{M}$ a computation from p to q popping the nonterminal A from the stack, or more precisely make the square occupied by A empty for the first time.

More formally, we prove for all $q, p \in Q$ and $A \in \Gamma$ that

$$[q, A, p] \Rightarrow^*_{\mathcal{G}} x \in \Sigma^* \qquad \text{iff} \qquad (q, x, A) \vdash^*_{\mathcal{M}} (p, 1, 1). \tag{3.15}$$

This is done by induction on the number i of steps in a derivation of $\mathcal{G}$ or in a computation of $\mathcal{M}$.

I. First we show that

$$\text{if} \quad (q, x, A) \vdash^i_{\mathcal{M}} (p, 1, 1), \quad \text{then} \quad [q, A, p] \Rightarrow^*_{\mathcal{G}} x.$$

$i = 1$: Now $(p, 1)$ is in $\delta(q, x, A)$, so that $\mathcal{G}$ contains a production $[q, A, p] \to x$.

Induction step: Consider a computation of $\mathcal{M}$ of length i, and write it in the form

$$(q, ay, A) \vdash (q_1, y, B_1 \cdots B_n) \vdash^{i-1} (p, 1, 1), \tag{3.16}$$

with $x = ay$, $a \in \Sigma \cup \{1\}$. Now, y has a factorization

$$y = y_1 \cdots y_n,$$

where y_j has the effect of popping B_j from a stack in the sense that during the shown $i - 1$ steps on the computation the square originally filled by B_j is made empty for the first time when reading y_j. Of course, this popping can be done after a long sequence of moves, where the stack can be longer than $n - j$, but no symbols $B_{j+1}, \ldots, B_n$ are

touched when reading y_j. Some y_j´s may be empty, or may contain "at the end several occurrences of the empty word".

More precisely y_j´s are defined such that for some states $q_2, \ldots, q_{n+1}$, with $q_{n+1} = p$, we have

$$(q_j, y_j, B_j) \vdash^*_{\mathcal{M}} (q_{j+1}, 1, 1).$$

All of these computations are shorter than i, so that the induction hypothesis applies yielding

$$[q_j, B_j, q_{j+1}] \Rightarrow^*_{\mathcal{G}} y_j \quad \text{for } 1 \leq j \leq n.$$

Now, by the construction of $\mathcal{G}$, we conclude from the first step of (3.16) that

$$[q, A, p] \Rightarrow_{\mathcal{G}} a[q_1, B_1, q_2] \cdots [q_n, B_n, q_{n+1}],$$

so that combining these derivations of $\mathcal{G}$ we obtain

$$[q, A, p] \Rightarrow^*_{\mathcal{G}} ay_1 \cdots y_n = ay = x,$$

as required.

II. Secondly we show that

$$\text{if} \quad [q, A, p] \Rightarrow^i_{\mathcal{G}} x \in \Sigma^*, \quad \text{then} \quad (q, x, A) \vdash^*_{\mathcal{M}} (p, 1, 1).$$

This again is proved by induction on i.

$i = 1$: Now $[q, A, p] \to x$ must be a production so that $(p, 1) \in \delta(q, x, A)$.

Induction step: consider a derivation of $\mathcal{G}$ of the length i of a terminal word x:

$$[q, A, p] \Rightarrow_{\mathcal{G}} a[q_1, B_1, q_2] \cdots [q_n, B_n, q_{n+1}] \Rightarrow^{i-1}_{\mathcal{G}} x, \tag{3.17}$$

where $q_{n+1} = p$. Then we can write $x = ax_1 \cdots x_n$, where

$$[q_j, B_j, q_{j+1}] \Rightarrow^*_{\mathcal{G}} x_j \quad \text{for } 1 \leq j \leq n.$$

Moreover, each of these derivations are shorter than i, so that the induction hypothesis applies yielding

$$(q_j, x_j, B_j) \vdash^*_{\mathcal{M}} (q_{j+1}, 1, 1) \quad \text{for } 1 \leq j \leq n.$$

In these computations we can add $B_{j+1} \cdots B_n$ to the bottom of the stack:

$$(q_j, x_j, B_j B_{j+1} \cdots B_n) \vdash^*_{\mathcal{M}} (q_{j+1}, 1, B_{j+1} \cdots B_n) \quad \text{for } i \leq j \leq n.$$

Now, from the first step of (3.17) we conclude that

$$(q, x, A) \vdash_{\mathcal{M}} (q_1, x_1 \cdots x_n, B_1 \cdots B_n),$$

and therefore combining the above computations we obtain

$$(q, x, A) \vdash^*_{\mathcal{M}} (p, 1, 1),$$

as required.

By now, we have established (3.15). If we choose $q = q_0$ and $A = z_0$, we obtain

$$[q_0, z_0, p] \Rightarrow^*_{\mathcal{G}} x \in \Sigma^* \qquad \text{iff} \qquad (q_0, x, z_0) \vdash^*_{\mathcal{M}} (p, 1, 1).$$

This together with the productions $S \to [q_0, z_0, p]$, for some $p \in Q$, implies that

$$S \Rightarrow^*_{\mathcal{G}} x \in \Sigma^* \qquad \text{iff} \qquad (q_0, x, z_0) \vdash^*_{\mathcal{M}} (p, 1, 1) \quad \text{for some } p \in Q.$$

Therefore $L(\mathcal{G}) = N(\mathcal{M})$, as was to be proved. □

We obtain from the first part of the above proof:

Corollary 3.22. *Each CF language is accepted with empty stack by a pda having only one state.*

Remark 3.10. Above corollary says that states are actually useless in pda´s, if the acceptance is with empty stack. This result is useful for certain theoretical considerations. However, in order to show that certain languages are CF it is very useful to have final states.

Theorem 3.21 can be used to show further closure properties of the family of CF languages.

Theorem 3.23. *For each CF language $L \subseteq \Sigma^*$ and regular language $R \subseteq \Sigma^*$ the language $L \cap R$ is CF.*

Proof. An intuitive reason for the result is clear: a finite memory of an FA added to a pushdown memory + a finite memory of a pda is still a pushdown memory + a finite memory. This is also the idea of the proof: FA $\mathcal{A}$ and pda $\mathcal{M}$ are run simultaneously on inputs which are accepted iff both $\mathcal{A}$ and $\mathcal{M}$ accept.

Formally, let $R = L(\mathcal{A})$ for DFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, q_0, F_\mathcal{A})$ and $L = L(\mathcal{M})$ for pda $\mathcal{M} = (Q_\mathcal{M}, \Sigma, \Gamma, \delta_\mathcal{M}, p_0, z_0, F_\mathcal{M})$. Define

$$\mathcal{M}_\cap = \big(Q_\mathcal{A} \times Q_\mathcal{M}, \Sigma, \Gamma, \delta, (q_0, p_0), z_0, F_\mathcal{A} \times F_\mathcal{M}\big),$$

where δ is defined as

$$\big((p', q'), \gamma\big) \in \delta\big((p, q), a, z\big) \qquad \text{iff} \qquad \delta_\mathcal{A}(p, a) = p' \quad \text{and} \quad (q', \gamma) \in \delta_\mathcal{M}(q, a, z).$$

Of course, a above is in $\Sigma \cup \{1\}$. If $a = 1$, then by the definition of $\delta_\mathcal{A}$, $\delta_\mathcal{A}(p, a) = p$.

We claim that for any $\gamma \in \Gamma^*$, $w \in \Sigma^*$ and $(p, q) \in Q_\mathcal{A} \times Q_\mathcal{M}$ we have

iff
$$\big((p_0, q_0), w, z_0\big) \vdash^i_{\mathcal{M}_\cap} \big((p, q), 1, \gamma\big)$$
$$\delta_\mathcal{A}(p_0, w) = p \quad \text{and} \quad (q_0, w, z_0) \vdash^i_\mathcal{M} (q, 1, \gamma).$$

This follows directly from the construction (and can be formally proved by induction).

Now, Theorem follows from the above equivalence. □

Theorem 3.24. *Let $L \subseteq \Delta^*$ be CF and $h : \Sigma^* \to \Delta^*$ a morphism. Then $h^{-1}(L) \subseteq \Sigma^*$ is CF.*

Proof. Let $L = L(\mathcal{M})$ for pda $\mathcal{M} = (Q, \Delta, \Gamma, \delta, q_0, z_0, F)$. We construct a pda $\mathcal{M}'$ such that:

– On input a (or w) $\mathcal{M}'$ behaves as $\mathcal{M}$ does on $h(a)$ (or $h(w)$).

A problem here is that a computation $(p, h(a), z) \vdash^*_\mathcal{M} (q, 1, \gamma)$ might contain unboundedly long γ — and hence cannot be simulated by $\mathcal{M}'$ on one (or a fixed finite number of) step(s). A solution to this is that $\mathcal{M}'$ remembers in its states (in a "buffer" of a finite length) the suffixes of $h(a)$´s.

Formally, $\mathcal{M}' = \big(Q', \Sigma \cup \Delta, \Gamma, \delta', (q_0, 1), z_0, F \times \{1\}\big)$, where

$$Q' = \{(q, \alpha) \mid q \in Q, \alpha \text{ is a suffix of } h(a) \text{ for some } a \in \Sigma\},$$

and δ' is defined as:

(i) $\big((p,\alpha),\gamma\big) \in \delta'\big((q,\alpha),1,z\big)$, if $(p,\gamma) \in \delta(q,1,z)$;

(ii) $\big((p,\alpha),\gamma\big) \in \delta'\big((q,a\alpha),1,z\big)$, if $(p,\gamma) \in \delta(q,a,z)$ for $a \in \Delta$; and

(iii) $\Big(\big(q,h(a)\big),z\Big) \in \delta'\big((q,1),a,z\big)$ for all $a \in \Sigma$, $z \in \Gamma$.

The meaning of (i)–(iii) are in that order: "Simulates a 1-move of $\mathcal{M}$"; "Simulates a move of $\mathcal{M}$ which reads $a \in \Delta$"; and "Loads a buffer by $h(a)$". In the first move the buffer is unchanged, while in the second move it is shortened by one.

We claim that $L(\mathcal{M}') = h^{-1}(L)$.

I. $h^{-1}(L) \subseteq L(\mathcal{M}')$. Now, if $\big(q,h(a),\beta\big) \vdash^*_{\mathcal{M}} (p,1,\beta')$, then one application of (iii) followed by several applications of (i) and (ii) shows that

$$\big((q,1),a,\beta\big) \vdash^*_{\mathcal{M}'} \big((p,1),1,\beta'\big).$$

Consequently,

$$\text{if} \quad \big(q_0,h(w),z_0\big) \vdash^*_{\mathcal{M}} (p,1,\gamma), \quad \text{then} \quad \big((q_0,1),w,z_0\big) \vdash^*_{\mathcal{M}'} \big((p,1),1,\gamma\big),$$

proving the inclusion $h^{-1}(L) \subseteq L(\mathcal{M}')$.

II. $L(\mathcal{M}') \subseteq h^{-1}(L)$. Let $w = a_1 \cdots a_n$, with $a_i \in \Sigma$, be accepted by $\mathcal{M}'$. By the construction, $\mathcal{M}'$ can read a symbol only in the states $(q,1)$, that is when the buffer is empty. Consequently, the accepting computation of w in $\mathcal{M}'$ is of the form

$$\begin{aligned}
\big((q_0,1),a_1\cdots a_n,z_0\big) &\vdash^*_{\mathcal{M}'} \big((p_1,1),a_1\cdots a_n,\gamma_1\big)\\
&\vdash^*_{\mathcal{M}'} \Big(\big(p_1,h(a_1)\big),a_2\cdots a_n,\gamma_1\Big)\\
&\vdash^*_{\mathcal{M}'} \big((p_2,1),a_2\cdots a_n,\gamma_2\big)\\
&\vdash^*_{\mathcal{M}'} \Big(\big(p_2,h(a_2)\big),a_3\cdots a_n,\gamma_2\Big)\\
&\qquad\vdots\\
&\vdash^*_{\mathcal{M}'} \Big(\big(p_n,h(a_n)\big),1,\gamma_n\Big)\\
&\vdash^*_{\mathcal{M}'} \big((p_{n+1},1),1,\gamma_{n+1}\big),
\end{aligned}$$

where $p_{n+1} \in F$. In above, transitions from state $(p_i,1)$ to state $\big(p_i,h(a_i)\big)$ is by (iii), while all the other transitions are by rules in (i) and (iii). The latter ones have counterparts in $\mathcal{M}$, so that in $\mathcal{M}$ we must have

$$(q_0,1,z_0) \vdash^*_{\mathcal{M}} (p_1,1,\gamma_1) \quad \text{and} \quad (p_i,h(a_i),\gamma_i) \vdash^*_{\mathcal{M}} (p_{i+1},1,\gamma_{i+1}),\ \forall i.$$

Consequently,

$$\big(q_0,h(a_1\cdots a_n),z_0\big) \vdash^* (p_{n+1},1,\gamma_{n+1}), \quad \text{with } p_{n+1} \in F.$$

This means that $h(w) \in L(\mathcal{M})$, and therefore $w \in h^{-1}(L)$. □

From Nivat´s Theorem (or its Corollary 2.26) and Theorems 3.6, 3.23 and 3.24 we obtain the following useful closure property.

Theorem 3.25. *The family of CF languages is closed under rational transductions, i.e. for each CF language $L \subseteq \Sigma^*$ and rational transduction $\mathcal{T} : \Sigma^* \to \Delta^*$, the language $\mathcal{T}(L)$ is CF.*

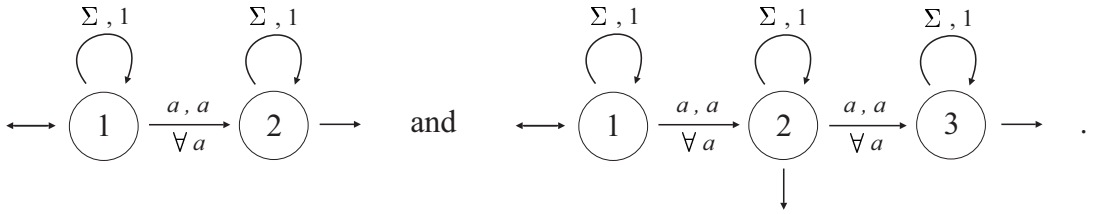
Example 3.8. For each CF language L the languages

$$F(L) = \{w \mid \exists u, v \in \Sigma^* : \ uwv \in L\}$$

and

$$F_2(L) = \{ww' \mid \exists u, v, z \in \Sigma^* : \ uwvw'z \in L\}$$

are CF. Indeed they are images of L under the finite transducers



3.4 Restrictions and extensions

In this section we consider briefly some subfamilies of the family of CF languages, as well as some extensions. A particularly interesting subfamily is the family of *deterministic CF languages*. Its importance follows from the fact that in most applications, when the theory of CF languages is applied to programming languages, it suffices to consider deterministic CF languages. These are defined via a deterministic version of pda.

Definition 3.11. Pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is *deterministic*, dpda for short, if the transition relation satisfies:

(i) $|\delta(p, a, z)| \leq 1$ for each $p \in Q$, $a \in \Sigma \cup \{1\}$ and $z \in \Gamma$, and

(ii) if $\delta(p, 1, z) \neq \emptyset$, then, for all $a \in \Sigma$, $\delta(p, a, z) = \emptyset$.

Further $L \subseteq \Sigma^*$ is *deterministic* CF language, that is in Det, iff there exists a dpda $\mathcal{M}$ such that $L = L(\mathcal{M})$, that is, is *accepted by a dpda with final states.*

It follows that in a dpda each input word has at most one computation (either accepting or not). Hence, dpda´s are unambiguous. Note also that, it is not natural to define the acceptance with the empty stack. If this would have been done, then in a deterministic CF language no proper prefix of an accepted word would be accepted (since the computation always halts if the stack is empty). In particular, Lemma 3.18 on page 55 does not hold for dpda´s. On the other hand, the construction of Lemma 3.19 on page 56 preserves the determinism.

Note also that in Example 3.7 on page 55 the first pda is deterministic, while the second is not.

Example 3.9. Language $\{a^n b^n c^m \mid n, m \geq 1\}$ is in Det. Indeed, it is accepted by a dpda:

$$\begin{aligned}
\delta(q_0, a, z_0) &= (q_0, az_0) \\
\delta(q_0, a, a) &= (q_0, aa) \\
\delta(q_0, b, a) &= (q_1, 1) \\
\delta(q_1, b, a) &= (q_1, 1) \\
\delta(q_1, c, z_0) &= (q_f, z_0) \\
\delta(q_f, c, z_0) &= (q_f, z_0),
\end{aligned}$$

with q_f final. Hence, as in the case of CF languages, Det is not closed under intersection.

In general, the closure properties of Det are quite different from those of $\mathcal{CF}$:

Theorem 3.26. Det *is closed under inverse morphisms and intersection with regular language.*

Proof. The constructions of the proofs of Theorems 3.24 and 3.25 preserve the determinism. □

In the next two propositions we state some differences of Det and $\mathcal{CF}$ with respect to closure properties.

Proposition 3.27. Det *is closed under complementation.*

Idea of proof. Make a dpda complete (as in the case of an FA) and change the final and nonfinal states. However, this is rather complicated (cf. Harrison), in general. Only if a dpda does not contain 1-transitions, then this can be done easily by introducing a new garbage state. □

Example 3.10. Language $L = \{a^n b^m c^k \mid n = m \text{ or } m = k\}$ is not in Det. Suppose the contrary. Then

$$L' = (\Sigma^* \setminus L) \cap a^* b^* c^* = \{a^n b^m c^k \mid n \neq m \text{ and } m \neq k\}$$

would be CF by Proposition 3.27 and Theorem 3.24. However, this is not the case by Iteration Theorem: Pump the word

$$a^{m+m!} b^m c^{m+m!},$$

where b´s are marked and m is the constant of the theorem. Details are left as an exercise.

Proposition 3.28. Det *is not closed under any rational operation or morphisms.*

Proof. Based on Example 3.10. Indeed, define $L_1 = \{a^n b^m c^k \mid n = m\}$ and $L_2 = \{a^n b^m c^k \mid m = k\}$. Then L_1 and L_2 are in Det, as in Example 3.9, while

$$L = L_1 \cup L_2 = \{a^n b^m c^k \mid n = m \text{ or } m = k\}$$

is not. Hence, the nonclosure under union follows.

If Det would be closed under catenation, then

$$d^*(dL_1 \cup L_2) \cap da^*b^*c^* = dL_1 \cup dL_2$$

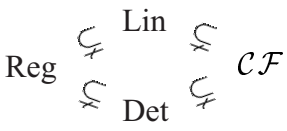
would be in Det, and so would be $L_1 \cup L_2$. Hence, the nonclosure under catenation is shown. The nonclosure under iteration is similar:

$$\big(\{d\} \cup (dL_1 \cup L_2)\big)^* \cap da^*b^*c^* = \{d\} \cup dL_1 \cup dL_2.$$

Finally, the nonclosure under morphisms follows since $dL_1 \cup eL_2 \in$ Det. □

Next we summarize our knowledge on hierarchy results of subfamilies of context–free languages.

Theorem 3.29. *We have*



and moreover the families Lin *and* Det *are incomparable.*

Proof. All the inclusions are clear by definitions. They are proper by languages $L_1 = \{a^nb^n \mid n \geq 0\}$, $L_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$ and $L_3 = \{a^nb^mc^k \mid n = m \text{ or } m = k\}$. Indeed, L_1 is not regular, L_2 is not linear and L_3 is not deterministic, while L_1 and L_2 are deterministic and L_1 and L_3 are linear, cf. Example 3.10 and Exercises. Therefore $L_3 \in$ Lin $\setminus$ Det and $L_2 \in$ Det $\setminus$ Lin showing the incomparability.

Note that our proof is based on Example 3.10 and therefore on Proposition 3.27. □

By Proposition 3.27 (or more precisely by its constructive proof) we obtain some new decidability results:

Theorem 3.30. *It is decidable, whether for a given regular language R and deterministic CF language L (i) $L = R$ and (ii) $R \subseteq L$.*

Proof. Now

$$L = R \qquad \text{iff} \qquad L_1 = \big(L \cap (\Sigma^* \setminus R)\big) \cup \big(R \cap (\Sigma^* \setminus L)\big) = \emptyset,$$

where L_1 is an effectively findable CF language by Proposition 3.27 and Theorems 3.26 and 3.8. Hence, the problem (i) is reduced to Theorem 3.16.

Problem (ii), in turn, follows from the identity $R \subseteq L$ iff $L \cap \Sigma^* \setminus R = \emptyset$. □

Remark 3.11. Some problems, as we shall see later, remain undecidable for the family Det. On the other hand, the *equivalence* problem of deterministic CF languages is often referred to as the most important problem of formal languages (which seems to have been solved 1997).

Remark 3.12. There exists a grammatical characterization of the family Det, using so–called *$LR(k)$-grammars*. These are very important on compiler constructions.

As in the case of finite automata we can extend pda´s by adding outputs:

Definition 3.12. A *pushdown transducer* $\mathcal{P}$, pdt for short, is an 8-tuple

$$\mathcal{P} = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, z_0, F),$$

where Δ is the output alphabet and $\delta \subseteq Q \times (\Sigma \cup \{1\}) \times \Gamma \times Q \times (\Delta \cup \{1\}) \times \Gamma^*$ is a finite set of transitions of the form

$$(p, u, z) \longrightarrow (q, v, \gamma),$$

where v is the output and the other components are as in a pda.

The notions of a pda, such as ID, are extended to pdt´s in a natural way. In particular pdt $\mathcal{P}$ computes a relation $R(\mathcal{P}) : \Sigma^* \to \Delta^*$ as follows:

$$R(\mathcal{P}) = \{(u, v) \in \Sigma^* \times \Delta^* \mid (q_0, u, 1, z_0) \vdash_{\mathcal{P}}^* (q, 1, v, \gamma), q \in F, \gamma \in \Gamma^*\}.$$

Hence, in ID´s the second component tells the input so far read and the third one the output so far produced. Note also that we have chosen the acceptance with final states. Relations computed by pdt´s are called *algebraic*.

Example 3.11. The relation (or transduction) defined by

$$\begin{aligned} (ab)^n &\longmapsto a^n b^n \\ x &\longmapsto \emptyset \quad \text{otherwise} \end{aligned}$$

can be computed by a pdt with transitions:

$$\begin{aligned} (q_0, a, z_0) &\longrightarrow (q_b, a, z_0) & \qquad (q_a, 1, b) &\longrightarrow (q_t, b, 1) \\ (q_b, b, z_0) &\longrightarrow (q_a, 1, bz_0) & \qquad (q_t, 1, b) &\longrightarrow (q_t, b, 1) \\ (q_a, a, b) &\longrightarrow (q_b, a, b) & \qquad (q_t, 1, z_0) &\longrightarrow (q_f, 1, 1) \\ (q_b, b, b) &\longrightarrow (q_a, 1, bb) & \qquad & \text{with } q_f \text{ final.} \end{aligned}$$

Nivat Theorem has a counterpart here.

Theorem 3.31. *A relation $\rho \subseteq \Sigma^* \times \Delta^*$ is algebraic iff it is of the form*

$$\rho = \big\{\big(h(w), g(w)\big) \mid w \in L\big\}, \tag{3.18}$$

where $h : \Theta^* \to \Sigma^*$ and $g : \Theta^* \to \Delta^*$ are morphisms and $L \subseteq \Theta^*$ is CF.

Proof. $\Rightarrow$: Let $\rho = R(\mathcal{P})$. We first note that the set of accepting computations $\mathrm{Comp}(\mathcal{P})$, i.e. those sequences of transitions of $\mathcal{P}$ which are accepting, is a CF language. Indeed, this is accepted by a pda $\mathcal{M}_{\mathcal{P}}$ with the transitions

$$\big(p, (p, u, z) \to (q, v, \gamma), z\big) \longrightarrow (q, \gamma) \text{ in } \mathcal{M}_{\mathcal{P}} \qquad \text{iff} \qquad (p, u, z) \longrightarrow (q, v, \gamma) \text{ in } \mathcal{P}.$$

Now, clearly $L(\mathcal{M}_{\mathcal{P}}) = \mathrm{Comp}(\mathcal{P})$. Hence Θ can be chosen to be the set of transitions of $\mathcal{P}$ and morphisms h and g are defined to pick up u and v, respectively. Note, that in fact $\mathrm{Comp}(\mathcal{P})$ is deterministic.

$\Leftarrow$: For a pda $\mathcal{M}$ accepting $L \subseteq \Theta^*$ and morphisms $h : \Theta^* \to \Sigma^*$ and $g : \Theta^* \to \Delta^*$ satisfying (3.18), we can define a pdt $\mathcal{P}$ by the condition

$$\big(p, h(u), z\big) \longrightarrow \big(q, g(u), \gamma\big) \text{ in } \mathcal{P} \qquad \text{iff} \qquad (p, u, z) \longrightarrow (q, \gamma) \text{ in } \mathcal{M}.$$

Note that here on the left hand side we have a computation and not a single transition. Now clearly $R(\mathcal{P}) = \rho$. □

As in the case of finite transducers we can rewrite Theorem 3.31 as

Corollary 3.32. *A relation $\rho \subseteq \Sigma^* \times \Delta^*$ is algebraic iff it can be written in the form*

$$\rho = g \circ \bigcap L \circ h^{-1},$$

where h and g are morphisms and L is a CF language.

Now, from the closure properties of CF languages we obtain a result which should be compared to Theorem 3.25.

Theorem 3.33. *If $R \in$ Reg and $\mathcal{P}$ is a pdt, then $\mathcal{P}(R)$ is CF.*

Proof. By above Corollary 3.32 $\mathcal{P}(R) = g\big(h^{-1}(R) \cap L\big)$ for some CF language L. □

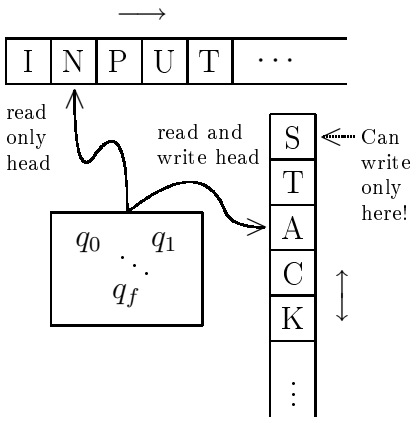
As our last example of this section we consider a generalization of a pda which can accept the language $\{a^n b^n c^n \mid n \geq 0\}$, for example.

Example 3.12. Let us call an extension of a pda a (1-way) *stack automaton* if it is like a pda in the sense that it consists of finite memory and one LIFO memory, which can be used by transitions of the form:

(i) a symbol can be pushed to the topmost square;

(ii) a symbol can be popped from the topmost square;

(iii) head of the stack can move one step to down or up *without writing* anything to the stack.



Moreover, in each step a symbol or empty word is read in the input tape and the state of the machine is allowed to change. Further endmarkers are used to identify the ends of the input.

Now, it is easy to describe a deterministic stack automaton SA accepting the language $\{a^n b^n c^n \mid n \geq 0\}$, or more precisely $\{\$a^n b^n c^n \# \mid n \geq 0\}$. When reading a´s it pushes those to the stack and stays in the topmost square. Then when encountering b SA moves to a nonwriting state and moves one step down in the stack when reading each b. When detecting the bottom marker z_0 of the stack the machine is able to continue iff it at the same time reads c. Now, when reading c´s the head in the stack moves one step upwards in each step and it accepts iff it reaches the topmost square at the same time, when the right endmarker of the input tape is scanned. Then and only then the machine enters to an accepting state.

Chapter 4

Context–sensitive languages

In this short chapter we consider the third family of languages in Chomsky hierarchy, namely the family of context–sensitive languages, $\mathcal{CS}$ in short. These are generated by context–sensitive grammars, cf. pages 6–7, where the productions are of the form

$$\alpha A\gamma \longrightarrow \alpha\beta\gamma \quad \text{with } A \in N,\ \beta \in V^+ \text{ and } \alpha, \gamma \in V^*, \tag{4.1}$$

or

$$S \longrightarrow 1,$$

where S is the start symbol, and if production $S \to 1$ appears in the grammar, then S does not occur on the right hand side of the productions. Productions of the form (4.1) are called *context dependent* productions.

Productions of (4.1) are also *length increasing* in the sense that the length of the right hand side is at least as large as that of the left hand side. Grammars having only length increasing productions are called *length increasing*. From this property we derive our first result.

Theorem 4.1. *The membership problem for CS languages is decidable.*

Proof. Let $\mathcal{G} = (V, \Sigma, S, \mathcal{P})$ be a CS grammar and $w \in \Sigma^*$. We have to construct an algorithm to decide whether "$w \in L(\mathcal{G})$?". The algorithm we present is very trivial (and inefficient):

(i) If $w = 1$, check whether $S \to 1 \in \mathcal{P}$, and if so output "yes";

(ii) Otherwise construct all nonrepetative sequences of words $w_0, w_1, \ldots, w_t$ over V such that $w = w_t$ and $|w_{i+1}| \geq |w_i|$ for $i = 0, \ldots, t-1$, and decide whether

$$S = w_0 \ \Rightarrow_{\mathcal{G}} \ w_1 \ \Rightarrow_{\mathcal{G}} \ \ldots \ \Rightarrow_{\mathcal{G}} \ w_i \ \Rightarrow_{\mathcal{G}} \ \ldots \ \Rightarrow_{\mathcal{G}} \ w_t = w. \tag{4.2}$$

If the sequence satisfying (4.2) is found output "yes", otherwise "no".

Clearly, in part (ii) there are only a finite number of sequences $(w_i)_{i \leq t}$ and for each such sequence (4.2) can be tested. Hence, the algorithm works correctly. □

As another very easy result we prove

Theorem 4.2. *The family of CS languages is closed under rational operations.*

Proof. We first note that for a given CS grammar we can effectively find another CS grammar such that the terminals occur only in the productions of the form $X \to a$, with $X \in N, a \in \Sigma$, cf. the proof of CNF for CF grammars.

Proof for iteration: Assume the normal form. Define a new starting symbol S_0, duplicate all nonterminals using banned letters, and duplicate also the productions accordingly. Finally add the productions:

$$\begin{aligned} S_0 &\longrightarrow 1 \mid \overline{S}S_1 \mid \overline{S} \\ S_1 &\longrightarrow \overline{\overline{S}}S_2 \mid \overline{\overline{S}} \\ S_2 &\longrightarrow \overline{S}S_1 \mid \overline{S} \end{aligned}$$

For other rational operations the Theorem follows directly from the corresponding proof for CF languages, when we assume that the nonterminal sets of different grammars are disjoint.

□

It follows directly from the definitions that for each CS language L the language $L \setminus \{1\}$ is length increasing, i.e. generated by such a grammar. Next we prove the converse which at the same time gives us better tools to show that some languages are CS.

Theorem 4.3. *Each language generated by a length increasing grammar is CS.*

Proof. First we show that for each CS grammar $\mathcal{G}_1 = (V, \Sigma, S, \mathcal{P})$, with $S \to 1 \notin \mathcal{P}$, and for each length increasing production

$$X_1 \cdots X_n = \alpha \longrightarrow \beta = Y_1 \cdots Y_m \quad \text{with } X_i, Y_i \in V \setminus \Sigma$$

the language generated by the grammar

$$\mathcal{G}_2 = (V, \Sigma, S, \mathcal{P} \cup \{\alpha \to \beta\})$$

is CS.

In order to prove this we define

$$\mathcal{G}_2' = (V \cup \{Z_1, \ldots, Z_n\}, \Sigma, S, \mathcal{P} \cup \mathcal{P}'),$$

where Z_i´s are new letters and $\mathcal{P}'$ consists of the productions:

$$\begin{aligned} X_1 \cdots X_n &\longrightarrow Z_1 X_2 \cdots X_n, \\ Z_1 X_2 \cdots X_n &\longrightarrow Z_1 Z_2 \cdots X_n, \\ &\vdots \\ Z_1 \cdots Z_{n-1} X_n &\longrightarrow Z_1 \cdots Z_n Y_{n+1} \cdots Y_m, \\ Z_1 \cdots Z_n Y_{n+1} \cdots Y_m &\longrightarrow Y_1 Z_2 \cdots Z_n Y_{n+1} \cdots Y_m, \\ &\vdots \\ Y_1 \cdots Y_{n-1} Z_n Y_{n+1} \cdots Y_m &\longrightarrow Y_1 \cdots Y_m. \end{aligned}$$

Clearly, $\mathcal{G}_2'$ is a CS grammar. It is also obvious that $L(\mathcal{G}_2) \subseteq L(\mathcal{G}_2')$. The inverse inclusion is also true, since if Z_i´s are introduced, they can be eliminated only by applying the whole list of the new productions, which is equivalent to use of the production $\alpha \to \beta$.

Now, we are ready to prove the Theorem. Let $\mathcal{G}$ be a length increasing grammar. By the argument of the proof of Theorem 4.2 we may assume that the terminals occur only in the productions of the form $X \to a$ with $X \in N$, $a \in \Sigma$. Now let $\mathcal{G}'$ be the CS grammar obtained from $\mathcal{G}$ by taking its all CS productions. Then, by the beginning of this proof, the language generated by a grammar which is obtained from $\mathcal{G}'$ by adding to it one of the productions of $\mathcal{G}$, which is not in $\mathcal{G}'$, is CS. Hence, by induction $L(\mathcal{G})$ is CS as well. □

Now we are able to give nontrivial examples of CS languages.

Example 4.1. Language $L = \{a^{2^n} \mid n \geq 1\}$ is CS. We gave a grammar for this language on page 7. It was not length increasing, but can be easily modified to be such. Indeed, replace the first production by

$$S \longrightarrow aa \mid aaaa \mid \#ta\#$$

third rules by

$$t\# \longrightarrow t'a\# \mid t'aa$$

to eliminate $t\# \longrightarrow t'$

and the last ones by

$$\#t' \longrightarrow aaaa \mid \#taa$$

in order to eliminate the production $\#t' \to 1$.

Theorem 4.4. $\mathcal{CF} \subsetneq \mathcal{CS}$.

Proof. Each CNF CF grammar is CS, implying the inclusion. By above example and Parikh Theorem, cf. also Example 3.5 on page 51, it is proper. □

Our next result points out a connection between the families of languages generated by CS and arbitrary grammars, respectively.

Theorem 4.5. *Let $L \subseteq \Sigma^*$ be a language generated by an arbitrary grammar, and a and b letters not in Σ. There exists a CS language L' such that*

(i) $L' \subseteq Lba^*$, and

(ii) for each $w \in L$ there exists an i such that $wba^i \in L'$.

Proof. Let $L = L(\mathcal{G})$ for a grammar $\mathcal{G} = (V, \Sigma, S, \mathcal{P})$. We define a length increasing grammar

$$\mathcal{G}' = (V \cup \{S', X, a, b\}, \Sigma \cup \{a, b\}, S', \mathcal{P}_1 \cup \mathcal{P}_2),$$

where $\mathcal{P}_1$ and $\mathcal{P}_2$ consists of the productions

$$\mathcal{P}_1 = \{\alpha \to \beta \mid \alpha \to \beta \in \mathcal{P} \text{ and } |\beta| \geq |\alpha|\} \cup \{\alpha \to \beta X^{|\alpha|-|\beta|} \mid \alpha \to \beta \in \mathcal{P} \text{ and } |\beta| < |\alpha|\}$$

and

$$\mathcal{P}_2 = \{S' \to Sb, bX \to ba\} \cup \{X\alpha \to \alpha X \mid \alpha \in V \cup \{b\}\}.$$

By Theorem 4.3, $L(\mathcal{G}')$ is CS. Moreover $L(\mathcal{G}') \subseteq Lba^*$, since forgetting letters X, a and b any derivation of $\mathcal{G}'$ is a derivation of $\mathcal{G}$, and the elimination of X´s can take place only on the right hand side of b which is introduced at the very beginning. Therefore (i) holds. Also (ii) holds, since any step of a derivation of $\mathcal{G}$ can be simulated by $\mathcal{G}'$ by one application of a production from $\mathcal{P}_1$ and a certain number of applications of productions in $\mathcal{P}_2$. □

From Theorem 4.5 we obtain

Corollary 4.6. *If there exists a language L which is generated by a grammar but which is not CS, then the family of CS languages is not closed under morphisms.*

Proof. Consider the morphism which is the identity on Σ and maps a and b to the empty word, and apply Theorem 4.5. □

Remark 4.1. It was proved only recently that the family of CS languages is closed under complementation (Immerman–Szelepscenyi Theorem).

Remark 4.2. As in the case of regular and context–free languages, there exists a class of automata, so called *linearly bounded automata*, lba for short, which accept exactly CS languages. From this class one can define *deterministic* CS languages — as languages accepted by deterministic variants of these automata. A big open question is whether these two families of languages coincide.

Chapter 5

Recursively enumerable languages

In this last chapter we consider the family of languages which is — as we shall see — the largest possible family of languages the elements of which are algorithmically defined. This family is defined via languages accepted by certain type of automata, so–called Turing machines. These machines play an important role in the history of computing, as well as the whole mathematics.

5.1 Turing machines

Turing machines were defined by Alan Turing in 1936 as a theoretical model for an algorithm, and they were important tools to formalize the notion of undecidability as well as to show that undecidable problems exist. This development in 1930´s destroyed the dream of Hilbert (from about year 1900) that all "well defined" mathematical problems are in principle algorithmically solvable.

The notion of a Turing machine can be defined in a number of different ways.

Definition 5.1. We define *Turing machine*, TM for short, as a seventuple

$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, *, F),$$

where

Q is the finite set of *states*,

Σ is the finite *input alphabet*,

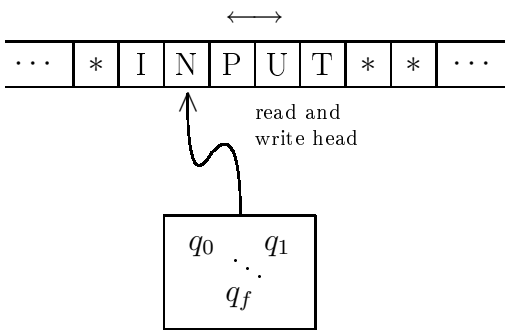
Γ is the finite tape alphabet with $\Sigma \subseteq \Gamma$,

δ is a (partial) *transition function*
$Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$,

$q_0 \in Q$ is the *initial state*,

$* \in \Gamma$ is the *blank symbol*,

$F \subseteq Q$ is the set of *final states*.



Consequently, a TM consists of a *finite control unit*, one *two–way infinite tape*, and one *head which is capable of reading and writing as well as moving to both directions.* Transitions are of the form

$$(p,a) \longrightarrow (q,b,X) \quad \text{with } p,q \in Q,\ a,b \in \Gamma \text{ and } X \in \{L,R\}$$

meaning that, when in state p the head is scanning a square containing a, the machine moves to the state q, replaces a by b and moves the head one square to the left or right.

Definition 5.2. An *instantaneous description* of $\mathcal{M}$, ID for short, is the word

$$\alpha_1 q \alpha_2 \in \Gamma^* Q \Gamma^+,$$

where $\alpha_1\alpha_2$ is the shortest contents of the tape containing the square pointed by the head and containing all squares filled by nonblank symbols. The *initial* ID is q_0w when w is the input. A *one step computation* or a *move* of $\mathcal{M}$ is defined as follows: Let $X_1 \cdots X_{i-1} p X_i X_{i+1} \cdots X_n$ be an ID and $\delta(p,X_i) = (q,Y,L)$ (resp. $\delta(p,X_i) = (q,Y,R)$) then we write

$$X_1 \cdots X_{i-1} p X_i \cdots X_n \ \vdash_{\mathcal{M}} \ \begin{cases} X_1 \cdots X_{i-2} q X_{i-1} Y X_i \cdots X_n & \text{if } i > 1 \\ q * Y X_i \cdots X_n & \text{if } i = 1. \end{cases}$$

$$\left(\text{resp.} \quad X_1 \cdots X_{i-1} p X_i \cdots X_n \ \vdash_{\mathcal{M}} \ \begin{cases} X_1 \cdots X_{i-1} Y q X_{i+1} \cdots X_n & \text{if } i < n \\ X_1 \cdots X_{i-1} Y q * & \text{if } i = n. \end{cases} \right)$$

Definition 5.3. Let $\vdash^*_{\mathcal{M}}$ (or $\vdash^*$ for short) be the reflexive and transitive closure of the above relation $\vdash_{\mathcal{M}}$. Then the language *accepted* by $\mathcal{M}$ is

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid q_0 w \vdash^*_{\mathcal{M}} uqv \quad \text{with } q \in F \text{ and } u,v \in \Gamma^*\}.$$

Hence, w is accepted if it causes a computation from the initial ID to an ID containing a final state.

For convenience *we assume that if a word is accepted, then the computation halts*, i.e. there is no next move. Hence, actually the set F could be $\{q\}$! If the word w is not accepted the computation might halt or might continue forever.

Definition 5.4. Next we associate two families of languages with TM´s. A language $L \subseteq \Sigma^*$ is *recursively enumerable* (or *type* 0), if there exists a TM $\mathcal{M}$ such that $L = L(\mathcal{M})$. Further a language $L \subseteq \Sigma^*$ is *recursive*, if there exists a TM $\mathcal{M}$ such that $L = L(\mathcal{M})$ *and* $\mathcal{M}$ *halts on all input words.* The corresponding families are denoted by RE and Rec.

The above definitions motivate a number of comments.

Remark 5.1. Our model of TM is deterministic. Several modifications and extensions, all equivalent to this basic model, are described later.

Remark 5.2. A TM differs from a 2FA only in the sense that it can write on the tape. Indeed, assuming that a 2FA does not go to a final state when moving to the left, which can be assumed, we observe that a language accepted by a 2FA is accepted by a TM (respecting our convention that TM always halts when accepting). Hence, by Proposition 2.21, Reg $\subseteq$ RE.

Remark 5.3. Always halting TM $\mathcal{M}$ provides an algorithm to decide whether an input word w is in $L(\mathcal{M})$, that is to solve the membership problem for $L(\mathcal{M})$. On the other hand, arbitrary TM $\mathcal{M}$ provides only procedure to confirm that $w \in L(\mathcal{M})$ if this is the case, but gives no information in the case $w \notin L(\mathcal{M})$. Hence, TM can be considered as a *semialgorithm* — it gives "yes" answers correctly.

Remark 5.4. TM $\mathcal{M}$ can be used to *compute (partial) algorithmic functions* $f_{\mathcal{M}} : \Sigma^* \to \Gamma^*$, for example, as follows:

$$f_{\mathcal{M}}(w) = u \qquad \text{iff} \qquad q_0 w \ \vdash_{\mathcal{M}} \ uqv \quad \text{with } q \in F \text{ and } u, v \in \Gamma^*.$$

Also in the case of an always halting machine $\mathcal{M}$, it can be used to solve the membership problem for L by requiring that it computes

$$\chi_L : \Sigma^* \to \{0,1\}, \quad \chi_L(w) = \begin{cases} 0 & \text{if } w \notin L \\ 1 & \text{otherwise.} \end{cases}$$

Remark 5.5. Probably the most subtle feature of the TM is that it is not required to halt always. This is essential, if we want to capture the notion of an intuitive algorithm, as we do. This is demonstrated in the next example.

Example 5.1 (Diagonalization principle). Let the sequence

$$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \ldots \tag{5.1}$$

contain all algorithms in some formalization. (Of course, a well defined formalization must provide a finite description for each algorithm, so that they can be put into a sequence). Here we assume that each $\mathcal{A}_i$ is always halting. We claim that (5.1) does not contain all intuitive algorithms. Assume the contrary and consider the intuitive algorithm:

$$w \longrightarrow \boxed{\text{Find } \mathcal{A}_w} \longrightarrow \boxed{\text{Compute } \mathcal{A}_w(w)} \longrightarrow \boxed{\text{Add 1}} \longrightarrow \mathcal{A}_w(w) + 1.$$

Here we have assumed that $\mathcal{A}_i$´s are defined to compute algorithmic functions $\mathbb{N} \to \mathbb{N}$. Let the index of the above algorithm in (5.1) be i_0. Then

$$\mathcal{A}_{i_0}(i_0) = \mathcal{A}_{i_0}(i_0) + 1,$$

a contradiction. The only assumptions needed for this contradiction are: the notion of an intuitive algorithm is formalized in such a way that the sequence (5.1) can be formed!

A way to avoid the above contradiction is to allow *nonhalting computations, even if we want to formalize only always halting algorithms.* In this light it becomes natural to allow (or even require) that TM´s need not halt in all of their inputs.

Remark 5.6. TM is more a theoretical tool rather than a practical method of constructing algorithms — as we shall see.

In what follows we give a number of examples, including several theoretical ones, showing the power of TM´s as language acceptors, as well as algorithms.

Example 5.2. $L = \{a^n b^n c^n \mid n \geq 1\}$ is accepted by the following TM $\mathcal{M}$:

$$(q_0, a) \longrightarrow (q_a, d, R)$$

$$\begin{array}{ll}
(q_a, a) \longrightarrow (q_a, a, R) & (\overleftarrow{q}, x) \longrightarrow (\overleftarrow{q_1}, x, L), \quad x \in \{a, b, c\} \\
(q_a, b) \longrightarrow (q_b, d, R) & (\overleftarrow{q_1}, x) \longrightarrow (\overleftarrow{q_1}, x, L), \quad x \in \{a, b, c, d\} \\
(q_b, b) \longrightarrow (q_b, b, R) & (\overleftarrow{q_1}, *) \longrightarrow (q_d, *, R) \\
(q_b, c) \longrightarrow (q_c, d, R) & (q_d, d) \longrightarrow (q_d, d, R) \\
(q_c, c) \longrightarrow (q_c, c, R) & (q_d, a) \longrightarrow (q_a, d, R) \\
(q_c, *) \longrightarrow (\overleftarrow{q}, *, L) & (q_a, d) \longrightarrow (q_a, d, R) \\
(\overleftarrow{q}, d) \longrightarrow (\overleftarrow{q}, d, L) & (q_b, d) \longrightarrow (q_b, d, R)
\end{array}$$

$$(\overleftarrow{q}, *) \longrightarrow (q_f, *, R).$$

Now, by the 7 first transitions, the machine checks that the input is in $a^+b^+c^+$, and simultaneously changes the first occurrence of each letter to d. When reaching the right end, i.e. when hitting to the blank, the machine goes to a new state $\overleftarrow{q}$, in which it travels through the tape and checks, whether it contains only d´s. If "yes" the word is accepted in state q_f. Otherwise, by using states $\overleftarrow{q_1}$ and q_d a new iteration is started, so that earlier written d´s are ignored, i.e. just passed.

It follows from the construction that $L = L(\mathcal{M})$. Clearly, the transition function here is only partial. However, it can be made complete by introducing a garbage state g, where the computation continues forever. *Note that this is true for any TM*! Further in above $\mathcal{M}$, $\Sigma = \{a, b, c\}$ and $\Gamma = \{a, b, c, d, *\}$.

Example 5.3. Here we point out several tasks which can be performed by TM´s, and which are useful as subroutines in TM constructions.

(i) *Marking the workspace*: Construct a TM such that

$$q_0' w \ \vdash^* \ \#q_0 w\#.$$

Clearly, such a machine is easy to construct assuming that w does not contain #.

(ii) *Finding a special symbol*: Construct a TM such that

$$q_0' w\#w' \ \vdash^* \ wq_0\#w', \qquad \text{with } \# \text{ not in } ww'.$$

(iii) *Making more space*: Construct a TM such that

$$q_0' w \ \vdash^* \ q_0 * w, \qquad \text{with } * \text{ not in } w.$$

(iv) *Copymachine*: Construct a TM such that

$$q_0' w \ \vdash^* \ q_0 w\#w, \qquad \text{with } * \text{ and } \# \text{ not in } w.$$

(v) *Comparison machine*: Construct a TM such that

$$q_0' w \# w' \vdash^* \begin{cases} w q_y \# w' & \text{if } w = w' \\ w q_n \# w' & \text{otherwise,} \end{cases}$$

where $\#$ and $*$ are not in ww'. Hence, making q_y final this accepts the language $\{w\#w \mid w \in \Sigma^+\}$.

For example the above Comparison machine can be constructed as follows:

- Mark the first symbol of w, say if a, change it to $\bar{a}$, and remember a in states;
- Search for $\#$ and move one step to the right;
- Compare a in the memory and the first symbol of w', if different go to state q_n after searching $\#$, if the same mark the second symbol of w'.
- Search for the marked symbol of w, and change its right neighbor marked and also put it to the memory of states;
- Search now the marked symbol of w', do the comparison of letters, change the marking one square to the right, and continue as in the previous search step.
- If the comparison can be done to the very ends of w and w', move to the state q_y after searching for $\#$.

It is obvious that each of the above steps can be realized by a certain TM, that is by a finite number of transitions. Hence, the comparison machine can be built.

Example 5.4 (Universal TM $\mathcal{M}_U$). We claim that there exists one fixed TM (that is a fixed finite number of deterministic transitions) which can simulate any TM $\mathcal{M}$. In other words,

$$\text{if} \quad q_0 w \vdash^*_{\mathcal{M}} u q_f v, \tag{5.2}$$

then $\mathcal{M}_U$ "does the same". How is this possible, since $\mathcal{M}$ may contain n input symbols, so that $\mathcal{M}_U$ would need infinite number of input symbols.

The answer is that we have to *encode* the input w of $\mathcal{M}$, as well as $\mathcal{M}$ itself into $\mathcal{M}_U$´s tape alphabet! Without loss of generality we assume that the tape alphabet Γ of $\mathcal{M}$ and the state set Q satisfy:

$$\Gamma \subseteq \{a_i \mid i \geq 0\} = \Sigma_a, \quad \text{with } * = a_0$$

and

$$Q \subseteq \{q_i \mid i \geq 0\} = \Sigma_q.$$

Further if $\mathcal{M}$ has n states, then $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ with q_0 initial and q_1 final. Note that we need only one final state since in final states computations stop.

Now we encode the words over Σ_a and Σ_q into the binary alphabet by a morphism $c : (\Sigma_a \cup \Sigma_q)^* \to \{0,1\}^*$,

$$\begin{aligned} a_i &\longmapsto 10^{2i+3}1 \\ q_i &\longmapsto 10^{2i+4}1. \end{aligned}$$

Further by defining $c(L) = 101$ and $c(R) = 1001$ we can encode the input $w = a_{i_1} \cdots a_{i_k}$ of $\mathcal{M}$ as

$$c(w) = c(a_{i_1}) \cdots c(a_{i_k}),$$

and a transition $t = (p, a, q, b, X)$ as

$$c(t) = c(p)\ c(a)\ c(q)\ c(b)\ c(X).$$

Finally, binary code of $\mathcal{M}$ containing exactly the transitions $t_1, \ldots, t_r$ is

$$c(\mathcal{M}) = 111\ c(t_1)\ 1\ c(t_2)\ 1\ \cdots\ 1\ c(t_r)\ 111.$$

Hence, $c(\mathcal{M})$ starts and ends with four 1´s and does not contain four consecutive 1´s inside. Note also that three consecutive 1´s in $c(\mathcal{M})$ show always a border between encodings of two transitions.

Now the universal $\mathcal{M}_U$ simulates $\mathcal{M}$, that is (5.2), in the sense

$$i\ c(\mathcal{M})\ c(w)\ \vdash^*_{\mathcal{M}_U}\ c(u)\ f\ c(v), \tag{5.3}$$

where i is the initial state of $\mathcal{M}_U$ and f its final state. The machine $\mathcal{M}_U$ operates as follows:

I. It *creates* to the beginning of its tape $c(q_0)c(a_{i_1})$, where a_{i_1} is the first letter of w, i.e.

$$i\ c(\mathcal{M})\ c(w)\ \vdash^*_{\mathcal{M}_U}\ \underset{\uparrow}{c}(q_0)\ c(a_{i_1})\ c(\mathcal{M})\ \overline{c(w)}, \tag{5.4}$$

where $\overline{c(w)}$ means that the first 1, that is a position of the head of $\mathcal{M}$ at the beginning of the computation is marked, and $\uparrow$ tells that the head of $\mathcal{M}_U$ is here in some state.

II. It *searches* for the transition used by $\mathcal{M}$ in the first step, and marks it by marking the occurrence of 1 just before this transition.

III. It *simulates* $\mathcal{M}$ by changing the state, that is the first block in the tape, and the marked symbol in the last block of the tape, and moreover moves the marked symbol in the last block according to the transition considered. It also removes the marking of the transition.

IV. It *changes* the second block of the tape to the symbol now marked in the last block.

V. It *checks* whether the contents of the first block corresponds to the accepting state q_1 of $\mathcal{M}$. If "yes" $\mathcal{M}_U$ erases the three first blocks and searches for the marked position in the fourth block and enters to its final state. Otherwise it *continues* in II with the current values of the first two blocks.

Above the blocks mean the four coded parts of the right hand side of (5.4). It follows from the construction that $\mathcal{M}_U$ simulates $\mathcal{M}$ in the sense of (5.3).

Therefore it remains to be shown that each step I–V can be realized by a finite number of (deterministic) transitions. In the quintuple level this is tedious. In the intuitive level it is not very difficult to become convinced of that:

In step I $\mathcal{M}_U$ has to create $c(a_{i_1})$ to the left on the input tape. This simply means that $\mathcal{M}_U$ has to search for the beginning of $c(w)$, which can be identified by 1^5, and copy from there a code of a_{i_1} to the left of the tape. And secondly create $c(q_0)$, i.e. 100001, still to the left of the above. Also the beginning of $c(w)$ must be marked. All

this can be realized by searching for a certain subword, using a suitable copymachine, marking a certain symbol and creating a fixed finite word to a given position.

The other steps also require only certain elementary operations, which can be achieved by certain finite sets of transitions. For example, in II the Comparison machine is needed to find out the transition in $c(\mathcal{M})$ to be used in the simulation. Further to replace a symbol or state (more precisely its coded version) requires a Comparison machine together with machine which creates a sufficient amount of space for the new symbol or state, cf. (iii) in Example 5.3.

All in all machines which can *search* for a marker, *mark* a new symbol, *copy*, *compare*, and *erase* (and some others) are enough to be used to construct $\mathcal{M}_U$.

As we already said the detailed construction according to the above lines would be tedious and boring. However, such a construction can be made, and the smallest known Universal Turing Machine´s are surprisingly small: There are such machines with

6 states and 6 tapesymbols

or

7 states and 4 tapesymbols.

Therefore, everything that can be done by TM´s can be done by only 28 transitions!! Here the final state is not counted.

Example 5.5 (Simulation of a many tape TM by ordinary one). As we saw in the construction of $\mathcal{M}_U$ above, the head of a TM might have to travel through the tape again and again. This can be avoided in some extend in many tape TM´s. A *k-tape* TM is like an ordinary TM except that instead of one tape only it contains k tapes, each of which is provided by an independent head. Further in each step the machine can

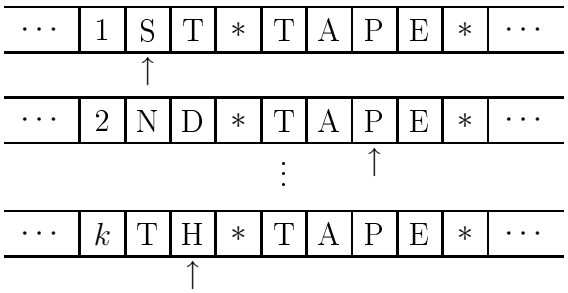
(i) change a state (which is common to all tapes);

(ii) write a new symbol on each tape to the square scanned by its head; and

(iii) move each of the head 0 or 1 steps to left or right.

Moreover, at the beginning the first tape contains the input, while the others contain only blank symbols $*$, and the word is accepted if the machine enters to a final state.

Now, we claim that any k-tape TM $\mathcal{M}$ can be simulated by an ordinary TM $\mathcal{M}'$ in the sense that $L(\mathcal{M}) = L(\mathcal{M}')$. Contrary to Example 5.4 no encoding is needed here. The simulation is very obvious: The ID of $\mathcal{M}$ can be illustrated as:



with common state q

Now in $\mathcal{M}'$ this is put into one tape having k tracks, and positions of the heads are indicated by marking one symbol in each track. Moreover, $\mathcal{M}'$ creates at the beginning endmarkers to tell the ends of the inputs. So the above ID corresponds in $\mathcal{M}'$ the ID:

$\dots$	#	1	$\overline{\text{S}}$	T	$*$	T	A	P	E	$*$	#	$*$	$\dots$
		2	N	D	$*$	T	A	$\overline{\text{P}}$	E	$*$		$*$	
		$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	
		k	T	$\overline{\text{H}}$	$*$	T	A	P	E	$*$		$*$	
			$\uparrow$										
			q										

The simulation of a step of $\mathcal{M}$ is performed by traveling in between endmarkers and changing at the same time each track according to the considered transition of $\mathcal{M}$. Note that formally transitions of $\mathcal{M}$ are of the form

$$\big(p, (a_1, \dots, a_k)\big) \longrightarrow \big(q, (b_1, X_1), \dots, (b_k, X_k)\big)$$

with $p, q \in Q$, $a_i, b_i \in \Gamma$ and $X_i \in \{L, R, 0\}$. Also the creation of endmarkers, as well as marking the initial positions of the heads to be the same as the position of the first head, can be done by suitable transitions. (To be precise when creating the endmarkers each letter a in the input w is changed to the vector $(a, *, *, \dots, *)^T$). Finally, $\mathcal{M}'$ accepts iff $\mathcal{M}$ does so.

It is clear, that such an $\mathcal{M}'$ can be constructed.

Example 5.6 (Simulation of a nondeterministic TM by ordinary one). A *nondeterministic* TM $\mathcal{M}$, NTM for short, is like a (deterministic) TM, but instead of a partial transition function it has a transition relation, i.e. a finite set of transitions

$$(p, a) \longrightarrow (q, b, X) \tag{5.5}$$

without assumption that, for each pair (p, a), there exists at most one triple (q, b, X) such that (5.5) is a transition. Hence, the computation caused by the input w is not unique, and w is *accepted*, that is $w \in L(\mathcal{M})$ iff *at least one* of these computations leads to a final state.

Now, we construct a deterministic three tape TM $\mathcal{M}'$ such that

$$L(\mathcal{M}') = L(\mathcal{M}).$$

Let d be the maximal number of right hand sides in (5.5) for any pair $(p, a) \in Q \times \Gamma$. Then any computation of $\mathcal{M}$ of length n is determined by a word of length n over the alphabet $\{1, \dots, d\}$: its ith letter tells which possibility in the ith step of the computation must be chosen (Hence, transitions for pairs (p, a) must be ordered).

The simulating machine $\mathcal{M}'$ contains three tapes:

The 1st tape contains the input of $\mathcal{M}$, and it is never changed;

The 2nd tape contains a word in $\{1, \dots, d\}^+$, more precisely all words in this alphabet are generated here in lexicographic order.

The 3rd tape is used to simulate a computation of $\mathcal{M}$ determined by a current contents of the second tape.

We need as a subroutine a TM $\mathcal{M}_g$ which computes

$$q_0\alpha \;\vdash^*\; q_0\, S(\alpha) \quad \text{for } \alpha \in \{1, \dots, d\}^+,$$

where $S(\alpha)$ is the next word to α in the lexicographic order. Such a machine is easy to construct.

Now, $\mathcal{M}'$ behaves as follows: After getting the input w of $\mathcal{M}$ it generates 1 to the second tape, copies w to the third tape and simulates on the third tape the computation of $\mathcal{M}$ determined by the word on the second tape. If $\mathcal{M}$ accepts so does $\mathcal{M}'$. Otherwise $\mathcal{M}'$ erases the word from the third tape and changes the word on the second tape to the next one in lexicographic order. Now, the same is repeated.

Intuitively, $\mathcal{M}'$ simulates in a row all computations of $\mathcal{M}$, and if any of those is accepting $\mathcal{M}'$ accepts. Hence $L(\mathcal{M}') = L(\mathcal{M})$, so that it remains to be convinced that $\mathcal{M}'$ can be constructed. But $\mathcal{M}'$ has to perform only certain simple tasks, which can be realized by a finite set of deterministic transitions.

Finally, the above $\mathcal{M}'$ can be replaced, by Example 5.5, by one tape deterministic TM.

Using any encoding, such as c on page 75, we can code any language $L \subseteq \Sigma^*$ to a language over $\{0,1\}$, such that we have one–to–one correspondence

$$\Sigma^* \supseteq L \ni w \longleftrightarrow c(w) \in c(L) \subseteq \{0,1\}^*.$$

Due to this, in many cases in formal language theory, the size of the alphabet is not important, if it is at least 2.

Our next result says that in TM´s additional tape symbols are not actually needed.

Proposition 5.1. *Each recursively enumerable language $L \subseteq \{0,1\}^*$ can be accepted by a TM with the tape alphabet $\Gamma = \{0,1,*\}$.*

The idea of the proof of Proposition 5.1 is to code tape symbols to the alphabet $\{0,1\}$, as in page 75. However, we omit the details here.

Remark 5.7. Consider a computation in a TM caused by an input w. It uses a certain number of steps as well as squares on the tape. These numbers are called the *time* and *space complexity* of this computation. This leads to important complexity classes of languages. Let $f : \mathbb{N} \to \mathbb{N}$ be a function. *Time* and *space complexity* classes associated with f are:

$$\begin{aligned}
\mathrm{TIME}(f) &= \{L \mid \exists \text{ TM } \mathcal{M} : \ L = L(\mathcal{M}) \quad \text{and} \quad \forall w \in L : \\
&\qquad w \text{ is accepted in at most } f(|w|) \text{ steps}\} \\
\mathrm{SPACE}(f) &= \{L \mid \exists \text{ TM } \mathcal{M} : \ L = L(\mathcal{M}) \quad \text{and} \quad \forall w \in L : \\
&\qquad w \text{ is accepted using at most } f(|w|) \text{ squares in the tape}\}.
\end{aligned}$$

5.2 Church´s thesis

Clearly, each Turing machine defines an intuitive (not necessary always halting) algorithm, i.e. a finitely defined effective procedure to associate an output to a given input.

How about reverse? This is claimed in so–called

Church´s Thesis. Turing machine is a formal counterpart of an intuitive notion of an algorithm.

This means that if something can be done by intuitively algorithmic procedure, it can be realized by a TM as well. Here this something can be:

– an algorithm to compute a function, for example, $\mathbb{N} \to \mathbb{N}$, $\Sigma^* \to \Delta^*$;

– an algorithm to decide a decision problem $w \to \boxed{\mathcal{A}} \to$ Yes/No;

– an effective procedure to list elements of a set.

Church´s Thesis, CT for short, *is not a mathematical statement in the sense that it could be proved true.* Indeed, what is an intuitive algorithm? How to show that TM can simulate any intuitive algorithm if we do not know what they are precisely?

In principle, it would be possible to disprove CT (if this would be the case), simply by introducing a problem which could be solved by an intuitive algorithm, and then proving that no TM can solve this problem. However, CT *is generally accepted* as an axiom type statement. The following facts strongly support this view:

1. No *counterexample* for CT has been found, i.e. no problem intuitively algorithmically solvable has been introduced, which could not be solved by a TM.

2. Strong *closure properties* of TM´s, i.e. all extensions of TM´s like those considered in Examples 5.5 and 5.6, lead to the same class of accepted languages or functions computed by these devices.

3. *Other formalizations*, such as recursive functions or grammars lead again to the same class of algorithmically computed functions as do TM´s.

5.3 Properties of recursively enumerable languages

In this section we consider some basic properties of recursively enumerable and also recursive languages. Moreover, a characterization, explaining the name "recursively enumerable", is given.

We start with

Theorem 5.2. *The family of recursive languages is closed under complementation.*

Proof. Let $L = L(\mathcal{M})$ for an always halting TM $\mathcal{M}$. We have to construct a TM $\mathcal{M}'$ which is always halting and accepts $\Sigma^* \setminus L$. Now, since $\mathcal{M}$ is deterministic and always halting, for each input w

(i) $\mathcal{M}$ halts in a final state accepting w, or

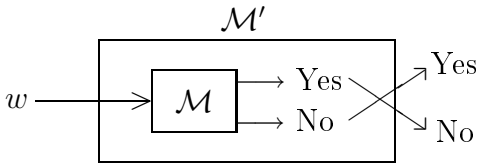
(ii) $\mathcal{M}$ halts in a nonfinal state rejecting w.

The halting means that there is no next move. Hence, we can make accepting computations rejecting by changing the final state of $\mathcal{M}$ to a nonfinal in $\mathcal{M}'$. Further each rejecting computation in $\mathcal{M}$ can be made accepting in $\mathcal{M}'$ by introducing a new final state f for $\mathcal{M}'$ and add to $\mathcal{M}'$ transitions

$$(p, a) \longrightarrow (f, a, R),$$

whenever in $\mathcal{M}$ there is no transition for the pair (p, a), with p nonfinal in $\mathcal{M}$. Otherwise $\mathcal{M}'$ is as $\mathcal{M}$.

Obviously, $L(\mathcal{M}') = \Sigma^* \setminus L(\mathcal{M})$. This construction can be illustrated as follows:



□

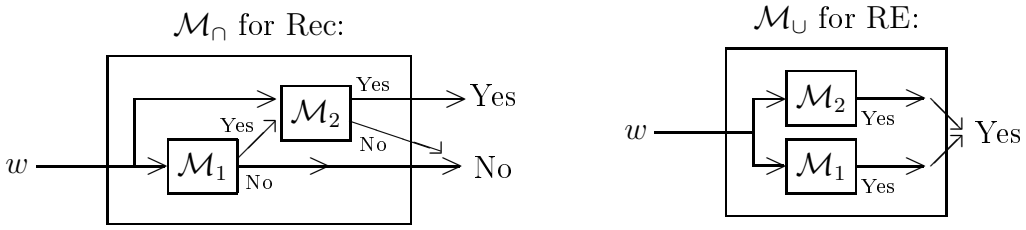
Theorem 5.3. *Both families* RE *and* Rec *are closed under union and intersection.*

Proof. Union.(i) Let L_1 and L_2 be recursive, i.e. $L_i = L(\mathcal{M}_i)$ for always halting TM´s $\mathcal{M}_1$ and $\mathcal{M}_2$. A two–tape TM $\mathcal{M}_\cup$ accepting $L_1 \cup L_2$ is easy to construct: For input w $\mathcal{M}_\cup$ first copies w to the second tape, then simulates $\mathcal{M}_1$ on the first tape and after halting $\mathcal{M}_2$ on the second tape. If either of the simulations is accepting, then $\mathcal{M}_\cup$ accepts. Clearly, $L(\mathcal{M}_\cup) = L_1 \cup L_2$, and since $\mathcal{M}_\cup$ is always halting, by Example 5.5, $L_1 \cup L_2 \in$ Rec.

(ii) If $L_i = L(\mathcal{M}_i)$, for $i = 1, 2$, for general TM´s $\mathcal{M}_1$ and $\mathcal{M}_2$, then the above does not work, since the first simulation need not halt. Now $\mathcal{M}_\cup$ is constructed as follows: It first copies the input w to the second tape, and then simulates alternately one step on the first and one step on the second tape and accepts if one of the simulations is accepting. This shows as above that $L_1 \cup L_2 \in$ RE.

Intersection. As above, except that the constructed $\mathcal{M}_\cap$ accepts iff both the simulations are accepting.

The above constructions can be illustrated as follows:



□

Our next result shows a connection between the families RE and Rec.

Theorem 5.4. *For any language $L \subseteq \Sigma^*$ we have:*

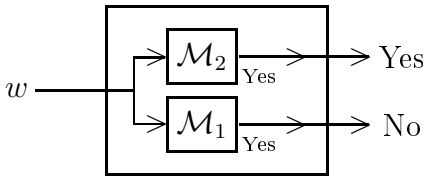
$$L \in \text{Rec} \qquad \Leftrightarrow \qquad L \ , \ \Sigma^* \setminus L \in \text{RE}\,.$$

Proof. $\Rightarrow$: Clear, by Theorem 5.2 and the fact that Rec $\subseteq$ RE by definitions.

$\Leftarrow$: Assume that $L = L(\mathcal{M}_1)$ and $\Sigma^* \setminus L = L(\mathcal{M}_2)$. We have to construct a TM $\mathcal{M}$ which always halts and accepts $L(\mathcal{M}_1)$. Again $\mathcal{M}$ will be originally a 2-tape TM (which is then converted to a 1-tape machine by Example 5.5). For a given input w, $\mathcal{M}$ first copies w to the second tape, then simulates alternately one step of $\mathcal{M}_1$ on the first tape and one step of $\mathcal{M}_2$ on the second tape. If the simulation on the first tape leads to a final state of $\mathcal{M}_1$, then $\mathcal{M}$ accepts (and halts). If the simulation on the second tape leads to a final state of $\mathcal{M}_2$, then $\mathcal{M}$ halts, but goes to a rejecting state. Exactly one of these cases takes place for any input, so that $\mathcal{M}$ always halts and accepts exactly $L(\mathcal{M}_1)$.

It is worth noting that although we know that exactly one of the above simulations for any w is accepting we do not know how many steps are needed for the acceptance.

Now, the illustration of the machine $\mathcal{M}$ is as follows:



□

It follows from Theorems 5.2 and 5.4 that we have the following exhaustive classification for pairs $(L, \Sigma^* \setminus L)$ of complementary languages:

(i) Both L and $\Sigma^* \setminus L$ are recursive;

(ii) Neither L nor $\Sigma^* \setminus L$ is recursively enumerable;

(iii) One of the languages L and $\Sigma^* \setminus L$ is recursively enumerable but not recursive, while the other is not recursively enumerable.

Next we prove that the inclusion Rec $\subseteq$ RE is proper. In doing so we consider the language

$$L_0 = \{c(\mathcal{M})\ c(w) \in \{0,1\}^* \mid \mathcal{M} \text{ is a TM and } w \in L(\mathcal{M})\},$$

where c is the encoding from page 75. We need one auxiliary result.

Lemma 5.5. *The language $L_M = \{c(\mathcal{M}) \mid \mathcal{M}$ is a TM$\}$ is recursive.*

Proof. By the definition of the encoding c

$$L_M = 111\big(1(00)^*0^411(00)^*0^311(00)^*0^411(00)^*0^311\{0,00\}1\big)^*111 \cap 111L_c111,$$

where

$$\begin{aligned} L_c = \{&w \in \big((10^+1)^5\big)^+ \mid \text{for each } i \text{ the number of zeros in the } (1+5i)\text{th block of zeros} \\ &\text{is different from 6, and for each } i \neq j \text{ the number of zeros in } (1+5i)\text{th and} \\ &(1+5j)\text{th or in } (2+5i)\text{th and } (2+5j)\text{th blocks of zeros are different}\}. \end{aligned}$$

The latter language guarantees that the TM is deterministic.

Clearly, the first part of the right hand side of L_M is regular, and hence recursive. The language $111L_c111$ can be accepted by an always halting TM, which is built from suitable comparing machines of Example 5.3. Hence, by Theorem 5.3, L_M is recursive. □

Theorem 5.6. $L_0 \in \text{RE} \setminus \text{Rec}$.

Proof. First we show that $L_0 \in$ RE. We construct a 3-tape TM $\mathcal{M}'$ accepting L_0.

First $\mathcal{M}'$ checks using the machine of Lemma 5.5 that a prefix of the input up to the second occurrence of four consecutive 1´s is in L_M. If "yes", then $\mathcal{M}'$ copies the suffix $c(w)$ to the second tape and the word $100001p$, where p is a prefix of w in 10^+1 to the third tape. Hence, the third tape contains a code of q_0 and that of the first letter of w.

From now on the simulation of $\mathcal{M}$ is as in the construction of Universal Turing Machine. The third tape contains the information about the current state and symbol scanned by the head, and based on it $\mathcal{M}'$ can find a transition from the first tape, and simulate it on the second tape, as well as change the contents of the third tape corresponding to the new state and new symbol scanned. After each simulation step

$\mathcal{M}'$ checks whether the state is final, i.e. the third tape starts as 10^61, and if "yes", then $\mathcal{M}$ accepts and so does $\mathcal{M}'$.

So we have proved that $L_0 \in \mathrm{RE}$.

In order to prove that $L_0 \notin \mathrm{Rec}$ we assume the contrary: $L_0 = L(\mathcal{M}')$ for an always halting TM $\mathcal{M}'$.

We claim that the language

$$L_d = \{c(\mathcal{M}) \mid \mathcal{M} \text{ is a TM } \textit{and} \text{ it accepts } c(\mathcal{M})\}$$

is recursive as well. An always halting TM $\mathcal{M}_d$ accepting L_d can be constructed as follows: First $\mathcal{M}_d$ checks that its input is in L_M and then changes it to the word $xc(x)$ simply copying (and coding it by c) to the end of the input (Here we have to assume that $0, 1 \in \Sigma_d$). Then $\mathcal{M}_d$ goes to the initial ID of $\mathcal{M}'$ on $xc(x)$ and continues as $\mathcal{M}'$ until it halts, and accepts iff $\mathcal{M}'$ accepts. So we have shown that $L_d \in \mathrm{Rec}$.

Next, by Theorem 5.2, also

$$L_2 = \Sigma^* \setminus L_d = \big\{v \in \{0,1\}^* \mid v \text{ is not a code of any TM } \textit{or} \\ \exists \text{ TM } \mathcal{M}: \ v = c(\mathcal{M}) \text{ and } \mathcal{M} \text{ does not accept } c(\mathcal{M})\big\}$$

is recursive, that is accepted by an always halting TM, say $\mathcal{M}_2$.

Consider now the word $v = c(\mathcal{M}_2)$:

$$c(\mathcal{M}_2) \in L_2 \underset{\text{def. of } L_2}{\Leftrightarrow} c(\mathcal{M}_2) \notin L(\mathcal{M}_2) \underset{\text{def. of } \mathcal{M}_2}{\Leftrightarrow} c(\mathcal{M}_2) \notin L_2,$$

a contradiction. Consequently, L_0 cannot be recursive. □

From above we derive:

Corollary 5.7. *The family* RE *is not closed under complementation.*

Proof. Theorems 5.2 and 5.6. □

Corollary 5.8. *The language* $L_d = \{c(\mathcal{M}) \mid \mathcal{M}$ *is a TM and* $\mathcal{M}$ *accepts* $c(\mathcal{M})\}$ *is recursively enumerable, but not recursive.*

Proof. By the proof of Theorem 5.6 $L_d \in \mathrm{RE}$. Further the contradiction in the same proof was derived from the recursiveness of L_d. □

More consequences of Theorem 5.6 are given in the next section.

We conclude this section by characterizing the family RE in such a way which motivates the term "recursively enumerable". Namely, we show that languages in RE are exactly those which can be *effectively listed.* Formally, this means that they can be listed by a Turing machine.

A Turing machine can be used as language generator as follows. The machine is a multitape machine having a special *output tape* on which the head can move only to the right and can write in each square only once. On this tape the words from Σ^* are written, words being separated by the marker #. At the beginning all the tapes are empty.

Let $\mathcal{M}$ be a TM defined above. The *language generated* by $\mathcal{M}$, $G(\mathcal{M})$ for short, is the set of all words $\mathcal{M}$ outputs in between markers #.

There are two obvious observations:

1. $G(\mathcal{M})$ is finite if $\mathcal{M}$ halts (when started at the blank input tapes);

2. If $L = G(\mathcal{M})$, $\mathcal{M}$ provides an effective procedure to list elements of L, but in which order is not known.

Now, we obtain a characterization:

Theorem 5.9. *For a language $L \subseteq \Sigma^*$ L is recursively enumerable iff L can be generated by a TM.*

Proof. $\Leftarrow$: Assume that $L = G(\mathcal{M})$ for a TM $\mathcal{M}$. We construct a TM $\mathcal{M}'$ as follows: $\mathcal{M}'$ has one tape more than $\mathcal{M}$, namely the input tape. For a given input w $\mathcal{M}'$ first simulates $\mathcal{M}$ on its all other tapes, and whenever $\mathcal{M}$ outputs #, $\mathcal{M}'$ tests whether its input w coincides with the word on the output tape immediately before the #. If "yes", then $\mathcal{M}'$ goes to a final state, and otherwise continues the simulation. Clearly, $L(\mathcal{M}') = G(\mathcal{M})$.

$\Rightarrow$: Let $L = L(\mathcal{M})$ for a TM $\mathcal{M}$. We have to construct a generator $\mathcal{M}'$ for L. This is not so obvious, but can be based on the idea:

"For each $i, j \geq 1$, $\mathcal{M}'$ simulates $\mathcal{M}$ i steps on the jth input word, and outputs the jth input, if this simulation is accepting". If such an $\mathcal{M}'$ can be constructed we are done: clearly, $\mathcal{M}'$ outputs only words from $L(\mathcal{M})$, and each word in $L(\mathcal{M})$ is produced by $\mathcal{M}'$ since it has an accepting computation of $\mathcal{M}$ of some finite length.

In order to construct $\mathcal{M}'$ we need TM´s for the following tasks:

(i) $\mathcal{M}_n$ which changes an input $w \in \Sigma^*$ to the next word $S(w)$ in the lexicographic order of Σ^*. Clearly, such a TM exists.

(ii) $\mathcal{M}_c$ which changes the pair $(i, j) \in \mathbb{N}_+^2$ (suitably encoded) to the next pair in the order $\leq$ defined as

$$(i, j) < (k, l) \quad \Leftrightarrow \quad i + j < k + l \quad \text{or} \quad i + j = k + l \ \text{ and } \ i < k.$$

We leave it as an exercise to conclude that this can be achieved by a TM.

Now, the required $\mathcal{M}'$ is constructed as follows: $\mathcal{M}'$ has five tapes including the output tape: the first one is to regenerate the input of $\mathcal{M}$, the second one to simulate $\mathcal{M}$ i steps on the jth input, the third and the fourth tape contain numbers i and j, and the fifth is the output tape.

The contents of the first tape is obtained by iterating the machine $\mathcal{M}_n$ as many times as shown by the contents of the fourth tape. Then this input is copied to the second tape where $\mathcal{M}$ is now simulated as many steps as the contents of the third tape indicates. If the simulation halts in the accepting state, then the contents of the first tape is produced in the fifth tape with marker #. Otherwise, the first and the second tape are erased and the pair (i, j) corresponding to contents of the third and fourth tape is replaced by the next pair. Now, the process is started again. □

5.4 Undecidability

In this section we come to one of the highlights of the theory of formal languages, or more precisely of computability. Namely, we show that there exist precisely defined, and even natural problems, which are algorithmically undecidable.

The existence of such problems is contrary to the general belief of mathematicians of the very beginning of this century. Indeed, D. Hilbert pointed out in International Congress of Mathematicians in 1900 a number of open problems, and one of those (so–called "*Hilbert´s 10th problem*") asked "to find a general algorithm which would decide whether a given Diophantine equation $P(x_1, \ldots, x_n) = 0$ has a solution in $\mathbb{Z}^n$". It was not thought at that time that the problem could have been undecidable, i.e. there does not exist any algorithm to solve it. This, however, turned out to be true (Matiyasevič, 1970).

How to show that a problem is algorithmically undecidable, that is no algorithm solves it? What is "any algorithm"? The notion of an algorithm has to be *formalized*, as a Turing machine for example, in order to be able to show that "something" is undecidable.

On the other hand, in order to show that "something" is decidable *no formalization is necessary*: it is enough to give an effective procedure, or an intuitive algorithm, which solves the problem.

Now, let us return to the language

$$L_0 = \{c(\mathcal{M})\ c(w) \mid \mathcal{M} \text{ is a TM and } w \in L(\mathcal{M})\}$$

of Theorem 5.6. It can be interpreted as a problem "Does a given TM $\mathcal{M}$ accept its input w?". Indeed, words in L_0 are coded forms of those instances of the problem, for which the answer is "yes". Words in $\{0,1\}^* \setminus L_0$, in turn, are those which correspond the "no" for the above problem, or are of the wrong form which also can be interpreted as "no" answer by considering an input word of the wrong form as a TM having no transitions and/or having its input not in Γ^*.

Definition 5.5. We call a problem *undecidable* iff there exists no always halting TM which solves it.

Of course, this requires that a problem has to be encoded to a suitable form for TM´s, i.e. to a language. It follows from CT that the above definition of the undecidability means that there is *no algorithm*, what so ever, to solve the considered problem.

It follows that Theorem 5.6 yields

Theorem 5.10. *The problem "Does a given TM $\mathcal{M}$ accept a given input w?" is undecidable, i.e. the corresponding language L_0 is not recursive.*

Similarly, from the Corollary 5.8 to Theorem 5.6 we obtain that the problem "Does a given TM $\mathcal{M}$ accept its own code $c(\mathcal{M})$?" is undecidable.

Once we have found one undecidable problem others can be obtained by a *reduction*: Let P_u be a known undecidable problem and P some other problem. If we can associate to an arbitrary instance i of P_u an instance $\varphi(i)$ of P such that

$$i \text{ is "yes"–instance} \qquad \text{iff} \qquad \varphi(i) \text{ is a "yes"–instance}$$

and the transformation $i \mapsto \varphi(i)$ can be done by a TM (algorithmically), then P is undecidable, as well. Indeed, to solve the P_u, its instance i can be transformed to $\varphi(i)$, and then apply a TM solving P, if such a TM would exist.

In terms of languages, i.e. coded instances of problems, we can formulate:

Definition 5.6. For languages $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$, L is *reduced* to L', in symbols $L \leq L'$, iff there exists a TM which computes a total function $\varphi : \Sigma^* \to \Delta^*$ such that

$$w \in L \qquad \text{iff} \qquad \varphi(w) \in L'. \tag{5.6}$$

It follows immediately, that if $L \leq L'$, then

(i) $L' \in \text{Rec} \;\Rightarrow\; L \in \text{Rec}$,

(ii) $L \notin \text{Rec} \;\Rightarrow\; L' \notin \text{Rec}$.

These conditions mean that if L and L' correspond to the codings of problems P and P', then (i) says that if P' is decidable so is P, and (ii) says that if P *is undecidable so is* P'.

Example 5.7. The problem P_h "Does $\mathcal{M}$ halt on its input w?" is undecidable. To see that we consider the language

$$L_h = \{c(\mathcal{M})\ c(w) \mid \mathcal{M} \text{ is a TM and } \mathcal{M} \text{ halts on } w\}.$$

We reduce L_0 to L_h, i.e. show that $L_0 \leq L_h$. Then the result follows from Theorem 5.6 and (ii) above.

Let $\mathcal{M}$ be a TM. Construct a TM $\mathcal{M}'$ such that it changes the halting, but rejecting computations to nonhalting ones. Further let φ be a mapping

$$\varphi:\ c(\mathcal{M})\ c(w) \;\mapsto\; c(\mathcal{M}')\ c(w).$$

Clearly, φ can be computed by a TM — it only has to add certain transitions to $\mathcal{M}$. Moreover, for any input w

$$\mathcal{M} \text{ accepts } w \qquad \text{iff} \qquad \mathcal{M}' \text{ halts on } w.$$

So indeed we have proved the reduction $L_0 \leq L_h$ and hence P_h is undecidable.

Example 5.8. The problem $P_{\exists h}$ "Does $\mathcal{M}$ halt on some of its inputs?" is also undecidable. The coded language is now

$$L_{\exists h} = \{c(\mathcal{M}) \mid \mathcal{M} \text{ is a TM and } \exists w \text{ such that } \mathcal{M} \text{ halts on } w\}.$$

Now P_h of Example 5.7 can be reduced to this as follows: For an instance of P_h, that is $(\mathcal{M}, w)$, we construct a TM $\mathcal{M}'$ such that

- $\mathcal{M}'$ forgets its own input x,
- generates the input w, and
- simulates $\mathcal{M}$ on w.

Such a $\mathcal{M}'$ can clearly be constructed. Now,

$\mathcal{M}$ halts on w iff $\mathcal{M}'$ halts on some input (iff $\mathcal{M}'$ halts on all of its inputs).

Hence, we have an informal proof that $P_{\exists h}$, as well as the problem $P_{\forall h}$ "Does $\mathcal{M}$ halt on all of its inputs?", is undecidable. To make it formal we should construct a TM computing

$$\varphi:\ c(\mathcal{M})\ c(w) \;\longmapsto\; c(\mathcal{M}').$$

Of course, this can be done.

Example 5.9. The language $L_{nh} = \{c(\mathcal{M}) \mid \mathcal{M} \text{ never halts}\}$ is not recursively enumerable. Indeed, if it would be, then so would be

$$\Sigma^* \setminus L_{\exists h} = L_{nh} \cup \{w \mid w \text{ is not a code of a TM}\} = L_{nh} \cup (\Sigma^* \setminus L_M).$$

Now, by the ideas of the proof of Theorem 5.9, $L_{\exists h}$ is in RE and by the previous example it is not in Rec. Hence $\Sigma^* \setminus L_{\exists h} \notin$ RE. Further $L_M \in$ Rec, and so is $\Sigma^* \setminus L_M$, implying, by Theorem 5.3, that also L_{nh} is not recursively enumerable.

Now we obtain easily

Theorem 5.11. *The equivalence problem for TM´s is undecidable.*

Proof. Let $\mathcal{M}_0$ be a TM which never halts. Then clearly, for any TM $\mathcal{M}$

$$L(\mathcal{M}) = L(\mathcal{M}_0) \qquad \text{iff} \qquad M_a \text{ never halts},$$

where $\mathcal{M}_a$ is obtained from $\mathcal{M}$ by making all halting but rejecting computations nonhalting. So the result follows from Example 5.9. (Formally the mapping φ is $c(\mathcal{M})c(\mathcal{M}_0) \mapsto c(\mathcal{M}_a)$). □

We have seen that a number of decision problems associated to TM´s are undecidable. Of course, all such problems are not so, for example "Does a TM have $2n$ states?" is clearly decidable. However, any problem asking "Does a given recursively enumerable language (defined by a TM) have a property P?" is *undecidable*, if only the property P is *nontrivial*, i.e. not true for all recursively enumerable languages!

Theorem 5.12 (Rice´s Theorem). *Each nontrivial property P of* RE *languages is undecidable, i.e. the language*

$$L_P = \{c(\mathcal{M}) \mid \mathcal{M} \textit{ is a TM and } L(\mathcal{M}) \textit{ has a property } P\}$$

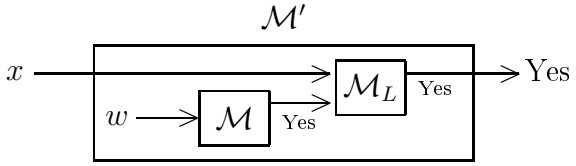
is nonrecursive.

Proof. We may assume that the empty language $L_\emptyset = \emptyset$ does not satisfy P — otherwise we consider the negation of P. On the other hand, there exists a TM $\mathcal{M}_L$ such that $L = L(\mathcal{M}_L)$ possesses the property P, i.e. $c(\mathcal{M}_L) \in L_P$.

We reduce the problem of Theorem 5.6 to P, that is we show that $L_0 \leq L_P$. In order to do so we associate to a pair $(\mathcal{M}, w)$ a TM $\mathcal{M}'$ as follows:

- After receiving an input x,
- $\mathcal{M}'$ first simulates $\mathcal{M}$ on w, and if this accepts,
- then simulates $\mathcal{M}_L$ on x, and $\mathcal{M}'$ accepts if $\mathcal{M}_L$ accepts.

This can be illustrated as:



Formally, we have to construct a TM which computes

$$c(\mathcal{M})\ c(w) \longmapsto c(\mathcal{M}'). \tag{5.7}$$

In order to do that we have to analyze what transitions are needed in $\mathcal{M}'$. They are those allowing to simulate $\mathcal{M}$ and $\mathcal{M}_L$, i.e. transitions of $\mathcal{M}$ and $\mathcal{M}_L$, and transitions allowed to print a w, as well as some which control the use of the above transitions. Since transitions of $\mathcal{M}_L$ are constant (independent of $c(\mathcal{M})c(w)$) they can be created. Also transitions printing w can be computed from $c(w)$. Hence, a TM computing (5.7) can be constructed: it creates a code of $\mathcal{M}'$ from $c(\mathcal{M})c(w)$.

By the construction of $\mathcal{M}'$, we have

$$\begin{aligned} w \in L(\mathcal{M}) &\Rightarrow L(\mathcal{M}') = L \\ w \notin L(\mathcal{M}) &\Rightarrow L(\mathcal{M}') = \emptyset, \end{aligned}$$

and therefore,

$$w \in L(\mathcal{M}) \quad \Leftrightarrow \quad L(\mathcal{M}') \text{ has property } P,$$

or more formally,

$$c(\mathcal{M})\, c(w) \in L_0 \quad \Leftrightarrow \quad c(\mathcal{M}') \in L_P.$$

□

Now, we turn to show that there are much more natural problems than the above ones which are undecidable. We already mentioned that Hilbert´s 10th problem is such. Another one, and particularly important from the point of view of formal languages, is the following problem.

Definition 5.7. *Post Correspondence Problem*, PCP for short, asks, for two given morphisms $h, g : \Sigma^* \to \Delta^*$, whether there exists a word $w \in \Sigma^+$ such that $h(w) = g(w)$, in other words, whether the *equality language* $E(h, g)$ of the pair (h, g) is nonempty:

$$E(h, g) = \{w \in \Sigma^+ \mid h(w) = g(w)\} \stackrel{?}{\neq} \emptyset. \tag{5.8}$$

We can fix $\Sigma = \{1, \ldots, n\}$. Then the *Modified Post Correspondence Problem*, MPCP for short, asks whether

$$E(h, g) \cap 1\Sigma^* \stackrel{?}{\neq} \emptyset. \tag{5.9}$$

Elements in (5.8) and (5.9) are called *solutions* of corresponding instances of PCP and MPCP.

We prove

Theorem 5.13 (Post, 1946). *PCP is undecidable.*

Proof. First we show that it is enough to prove that MPCP is undecidable:

Claim. MPCP reduces to PCP.

Proof. To prove this we associate with an arbitrary instance of MPCP, say

$$h : i \mapsto \alpha_i, \quad g : i \mapsto \beta_i, \quad \text{for } i = 1, \ldots, n,\ \alpha_i, \beta_i \in \Delta^*,$$

an instance of PCP $h', g' : \{0, 1, \ldots, n+1\}^* \to (\Delta \cup \{\#, \$\})^*$ such that

$$E(h, g) \cap 1\Sigma^* \neq \emptyset \quad \text{iff} \quad E(h', g') \neq \emptyset.$$

In order to define h' and g', we associate to a word $\gamma = a_1 \cdots a_n$, $a_i \in \Delta$, two new words

$$l(\gamma) = \#a_1\#a_2 \cdots \#a_n \quad \text{and} \quad r(\gamma) = a_1\#a_2\# \cdots a_n\#.$$

In particular for the empty word ϵ $\quad l(\epsilon) = r(\epsilon) = \epsilon$. Now, h' and g' are defined by

$$h': \begin{array}{rcl} 0 & \longmapsto & \#r(\alpha_1) \\ i & \longmapsto & r(\alpha_i) \\ n+1 & \longmapsto & \$ \end{array} \qquad g': \begin{array}{rcl} 0 & \longmapsto & l(\beta_1) \\ i & \longmapsto & l(\beta_i) \\ n+1 & \longmapsto & \#\$. \end{array} \quad \text{, for } i = 1, \ldots, n$$

Now assume that $w = 1w'$ is a solution of the instance (h, g) of MPCP. Then $h(1w') = g(1w')$, so that $\#r\big(h(1w')\big)\$ = l\big(g(1w')\big)\#\$$, which means that $h'(0w'\ n{+}1) = g'(0w'\ \mathrm{n}{+}1)$. Hence, $0w'\ n+1$ is a solution of (h', g').

Conversely, if w is a solution of the instance (h', g') of PCP, then w starts with 0 and ends with $n+1$, i.e. $w = 0w'\ n+1$. Moreover, we may assume, since the occurrences of \$ have to match in the h'- and g'-images of solutions, that w' does not contain either 0 or $n+1$. Therefore we obtain from the identity $h'(0w'\ n+1) = g'(0w'\ n+1)$ the identity $h(1w') = g(1w')$ simply erasing the markers. Hence, $1w'$ is a solution of (h, g).

Proof of Theorem now continues as follows. By Claim it is enough to reduce some undecidable problem P to MPCP. We use as P the problem of Theorem 5.10: "Decide whether a TM $\mathcal{M}$ accepts an input w".

We associate a pair $(\mathcal{M}, w)$ with an instance (h, g) of MPCP as follows: The morphisms $h, g : \Sigma^* \to \Delta^*$ are given in the following table, which at the same time fixes the alphabets Σ and Δ:

	$a \in \Sigma$:	$h(a)$:	$g(a)$:	
(0)	1	$\#$	$\#q_0w\#$	
(i)	X	X	X	, for $X \in \Gamma \cup \{\#\}$
(ii)	$(q, X) \longrightarrow (p, Y, R)$	qX	Yp	
	$\big(Z, (q, X) \longrightarrow (p, Y, L)\big)$	ZqX	pZY	
	$\big(\#, (q, X) \longrightarrow (p, Y, L)\big)$	$\#qX$	$\#p * Y$	
	$\big(\#, (q, *) \longrightarrow (p, Y, R)\big)$	$q\#$	$Yp\#$	
	$\big(\#, Z, (q, *) \longrightarrow (p, Y, L)\big)$	$Zq\#$	$pZY\#$	
(iii)	(X, Y, q)	XqY	q	, for $q \in F$
	(X, q)	Xq	q	
	(q, Y)	Yq	q	
(iv)	q	$q\#\#$	$\#$	, for $q \in F$,

where $\#$ is a marker, X, Y in (iii) and Z in (ii) ranges over the tape alphabet Γ, and letters or last components of letters in (ii) are transitions of $\mathcal{M}$. As usual we assume that there are no transitions from final states. Further, without loss of generality we can assume that the first move of $\mathcal{M}$ is to the right. If this is not the case originally, we introduce two extra moves to the beginning.

We have to show:

$$w \in L(\mathcal{M}) \qquad \text{iff} \qquad \text{MPCP } (h, g) \text{ has a solution in } 1\Sigma^*.$$

Assume first that $w \in L(\mathcal{M})$, i.e. in $\mathcal{M}$ there exists an accepting computation

$$q_0w \ \vdash\ \alpha_1q_1\beta_1 \ \vdash\ \alpha_2q_2\beta_2 \ \vdash\ \ldots \ \vdash\ \alpha_kq\beta_k, \quad \text{with } q \in F.$$

Then we can construct a solution δ for the MPCP as shown in the figure below:

$$
\begin{array}{rl}
\delta = & 1 \ \cdots \\
g: & \downarrow \quad \downarrow \ \downarrow \ \downarrow \\
& \overline{\underline{\#}\ \underline{q_0}\ \underline{w}\ \underline{\#}}\ \overline{\alpha_1\ q_1\ \beta_1}\ \overline{\underline{\#}}\ \alpha_2\ q_2\ \beta_2\ \#\ \cdots\ \underline{\#}\ \underline{\alpha_k}\ \underline{q}\ \underline{\beta_k}\ \overline{\underline{\#}}\ \overline{\alpha'_k\ q\ \beta_k}\ \overline{\#}\ \cdots\ \underline{\#}\ q\ \overline{\underline{\#}\ \underline{\#}} \\
h: & \uparrow\ \uparrow\ \uparrow\ \uparrow \\
\delta = & 1 \ \cdots
\end{array}
$$

Indeed, δ starts with 1, so that we have

$$
\begin{array}{c}
g(1) \\
\overline{\underline{\#}\ q_0\ w\ \#}. \\
h(1)
\end{array}
$$

Now q_0w *can be covered* by h-images, when by the construction, g-images define the next ID, namely $\alpha_1 q \beta_1$, and we obtain

$$
\begin{array}{ll}
g: & \overline{\underline{\#\ q_0\ w}\ \#\ \alpha_1\ q_1\ \beta_1}. \\
h: &
\end{array}
\tag{5.10}
$$

Now, if α_1 is empty and the second move is to the left, we take an h-image from the third alternate of (ii), and otherwise from (i), and we can cover $\#\alpha_1 q_1 \beta_1$ by an h-image, creating at the same time as a g-image the next ID of $\mathcal{M}$ $\alpha_2 q_2 \beta_2$ (possibly containing unnecessary blanks at the beginning and missing the last letter if it is $*$). By the same argument we obtain:

$$
\begin{array}{ll}
g: & \overline{\underline{\#\ q_0\ w\ \#\cdots\#}\ \alpha_k\ q\ \beta_k}\ \#. \\
h: &
\end{array}
\tag{5.11}
$$

Now, taking h-images from (i) and (iii) we can "erase α_k and β_k" step by step and obtain:

$$
\begin{array}{ll}
g: & \overline{\underline{\#\ q_0\ q\ \#\ \cdots}\ \underline{\#\ \alpha_k\ q\ \beta_k}}\ \overline{\cdots\ \underline{\#}\, q\ \#}\#. \\
h: &
\end{array}
$$

Hence, one application of (iv) completes the construction of δ.

Conversely, if the pair has a solution, which we can assume to be such that it is not a prefix of any other solution, then it is of the above form, showing that $w \in L(\mathcal{M})$. Indeed, any solution δ must start by 1, so that

$$
\begin{array}{ll}
g: & \overline{\underline{\#}\ q_0\ w\ \#}. \\
h: &
\end{array}
$$

Now, q_0w *can be covered* by h-images *only* in the way described earlier. This is so since $\mathcal{M}$ is deterministic. Therefore our solution δ satisfies (5.10) and we can continue with the same argument as long as a final state is not introduced in g-images. However, this has to happen, since otherwise the h-image would always be shorter than the g-image. So we have to come to a situation (5.11), proving that $w \in L(\mathcal{M})$. □

Next we give a number of applications of the undecidability of the PCP.

Example 5.10. Consider the following problem: Given a finite set of 3×3 matrices over natural numbers. Does there exist a product of these matrices having the same number in entries $(1,2)$ and $(1,3)$? This problem can be seen undecidable as follows. We associate a pair of morphisms $h, g : \Sigma^* \to \{2,3\}^*$ with a finite set of matrices

$$
M_a = \begin{pmatrix} 1 & h(a) & g(a) \\ 0 & 10^{|h(a)|} & 0 \\ 0 & 0 & 10^{|g(a)|} \end{pmatrix}, \quad a \in \Sigma,
$$

and observe that

$$M_a \cdot M_b = \begin{pmatrix} 1 & h(b)+h(a)\cdot 10^{|h(b)|} & g(b)+g(a)\cdot 10^{|g(b)|} \\ 0 & 10^{|h(a)|+|h(b)|} & 0 \\ 0 & 0 & 10^{|g(a)|+|g(b)|} \end{pmatrix} = \begin{pmatrix} 1 & h(ab) & g(ab) \\ 0 & 10^{|h(ab)|} & 0 \\ 0 & 0 & 10^{|g(ab)|} \end{pmatrix}.$$

Extending this formula for arbitrary products, we see that, if we could solve our problem we could solve also PCP in the case Δ is binary. But this is no restriction since images of letters can always be coded into a binary alphabet without changing the equality language, for example by using the encoding of page 75.

Next we move to undecidability results in formal language theory. We associate with an instance of PCP $h, g : \Sigma^* \to \Delta^*$, $\Sigma \cap \Delta = \emptyset$,

$$h: \quad i \longmapsto \alpha_i, \qquad g: \quad i \longmapsto \beta_i,$$

two languages

$$L_h = \{h(w)\#w^R \mid w \in \Sigma^+\} \quad \text{and} \quad L_g = \{g(w)\#w^R \mid w \in \Sigma^+\},$$

with $\# \notin \Sigma \cup \Delta$. Clearly, L_h and L_g are linear CF languages, for example L_h is generated by the grammar

$$\mathcal{G}_h: \quad S_h \longrightarrow h(i)S_h i \mid h(i)\#i, \quad \text{for } i \in \Sigma.$$

Further the CF grammar $\mathcal{G}_{amb}$ containing in addition to productions of $\mathcal{G}_h$ and $\mathcal{G}_g$ the productions $S \to S_h \mid S_g$ generates $L_h \cup L_g$. It follows that

$$L(\mathcal{G}_h) \cap L(\mathcal{G}_g) = \emptyset \qquad \text{iff} \qquad (h, g) \text{ has a solution} \qquad \text{iff} \qquad \mathcal{G}_{amb} \text{ is ambiguous.}$$

Hence, we obtain:

Theorem 5.14. *The following problems are undecidable:*

(i) Is the intersection of two CF languages (given by grammars) empty?

(ii) Is a given CF grammar ambiguous?

We conclude with another important undecidability result.

Theorem 5.15. *The equivalence problem for CF grammars is undecidable.*

Proof. We reduce the emptiness problem of Turing machines to the equivalence problem of CF grammars. The emptiness problem, i.e. the question "$L(\mathcal{M}) \stackrel{?}{=} \emptyset$", is undecidable, cf. Example 5.9 or Theorem 5.12.

With a TM $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, *, F)$ we associate a language

$$L_{nc} = \{w_1\#w_2^R\#w_3\#w_4^R\#\cdots\#w_n^\rho\# \mid n \geq 1, w_i \in (\Gamma \cup Q)^*, \text{ the sequence } w_1, \ldots, w_n$$
$$\text{is not a sequence of configurations of } \mathcal{M} \text{ in an accepting computation}\} \cup ((\Gamma^* Q \Gamma^* \#)^+)^C,$$

where $\# \notin \Gamma \cup Q$, $\Gamma \cap Q = \emptyset$, $w_n^\rho = w_n^R$ if $2 \mid n$, and $w_n^\rho = w_n$ if $2 \nmid n$.

Obviously

$$L(\mathcal{M}) = \emptyset \qquad \text{iff} \qquad L_{nc} = (\Gamma \cup \{\#\} \cup Q)^*.$$

Hence, the result follows if we can show that L_{nc} is CF. In fact, even a stronger result, namely the undecidability of the problem, whether $\mathcal{G}$ *generates all words over its input alphabet*, follows.

The CF-ness of L_{nc} is seen as follows. Clearly, $w \in L_{nc}$ iff (at least) one of the following conditions is true:

(i) w is of a wrong form, $w \notin (\Gamma^* Q \Gamma^* \#)^+$;

(ii) w_1 is not the initial ID, $w_1 \notin q_0 \Sigma^*$;

(iii) w_n is not an accepting ID, $w_n \notin \Gamma^* F \Gamma^+$;

(iv) for some odd i the ith word x_i and the $(i+1)$th word x_{i+1} in between markers satisfy $x_i \nvdash_{\mathcal{M}} x_{i+1}^R$;

(v) for some even i the ith word x_i and the $(i+1)$th word x_{i+1} in between markers satisfy $x_i^R \nvdash_{\mathcal{M}} x_{i+1}$.

Languages corresponding to conditions (i)–(iii) are regular. Moreover, languages corresponding to (iv) and (v) are context–free. Indeed, a pda accepting words satisfying (iv) can be constructed as follows. When reading an input, it nondeterministically guesses an odd i, and when reading x_i pushes into the stack x_i letter by letter except that the *only* occurrence of q with its neighbors, when encountered, is changed according to the transitions of $\mathcal{M}$, and then when reading x_{i+1} compares it to the contents of the stack. If they coincide, the pda accepts.

It follows that L_{nc} is a union of three regular languages and two CF languages, proving the Theorem. □

We conclude this section with a general remark.

Remark 5.8. We have been considering the four families of languages in Chomsky Hierarchy, namely Reg, $\mathcal{CF}$, $\mathcal{CS}$ and RE. From the point of view of decidability questions the following can be said:

(i) All problems for Reg are decidable;

(ii) Some problems for $\mathcal{CF}$ are decidable and some are not;

(iii) Almost all problems for $\mathcal{CS}$ are undecidable;

(iv) All problems for RE are undecidable.

We have proved several results, like Theorems 2.8, 3.16, 5.15, 4.1, 4.5, 5.12, supporting this view. However, for example (i) is not exactly true. Still the conditions (i)–(iv) provide useful hints of the decidability of problems in formal language theory.

5.5 Characterizations

We conclude this course by stating without proofs grammatical characterizations of RE and CS languages.

Proposition 5.16. *A language $L \subseteq \Sigma^*$ is recursively enumerable iff it is generated by a grammar.*

For CS languages the characterization points out an important complexity class, cf. page 79.

Proposition 5.17. *A language $L \subseteq \Sigma^*$ is context–sensitive iff it is accepted by a TM with endmarkers using no more space than the input requires.*

Actually the use of endmarkers in Proposition 5.17 is only for the clarity of the proof.