Research Group on Mathematical Linguistics
Rovira i Virgili University
Report GRLMC 21/01

Elements of Formal Language Theory for Membrane Computing

Carlos MARTÍN-VIDE

Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
E-mail: cmv@astor.urv.es

Gheorghe PĂUN[1]

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
E-mail: gpaun@imar.ro

Abstract. We introduce the basic elements of formal language theory which constitute the prerequisites for researches in membrane computing (P systems): basic notions and notations, Parikh mapping, semilinearity, Chomsky hierarchy, closure properties, decidability, descriptional complexity, normal forms, matrix grammars, random context grammars, Lindenmayer systems, the splicing operation, insertion-deletion operations, contextual grammars, grammar systems, finite automata, Turing machines, register machines. One gives rigorous definitions, most of them illustrated by examples, one recalls results of interest for membrane computing, one also formulates language theory open problems with recent motivation from membrane computing.

The collected items are classified in three classes: **B** = basics, **A** = advanced, **R** = related (elements marked with **R** were not yet used in membrane computing area, but it is highly possible that they will be useful for future researches).

[1]Work supported under the Programa Cátedra of the Fundacíon Banco Bilbao Vizcaya Argentaria (BBVA)

1 Introduction

This paper is a quick introduction to formal language theory (for a mathematically trained reader), collecting the notions and results from this area which are necessary (and, for the present stage, sufficient) for membrane computing. The need for such an introduction steamed from two observations. First, membrane computing can be considered as a branch of formal language theory, because, in spite of the fact that most of the variants of P systems deal with multisets of symbol-objects and compute sets of numbers or of vectors of numbers, the proof techniques, the generative style, the rewriting-type of multiset processing rules are very much similar to formal language theory. As a consequence, most papers about P systems start by long recollections of notions, notations, and results from formal language theory. Of course, for the sake of completeness, this cannot be (yet) avoided, but some uniformity in notations, for example, would be useful for the readers of membrane computing papers (if not also for their authors). Second, several times the authors of the present text have met people interested in membrane computing which expressed either their difficulty or their fear in entering this domain, because of the too specialised prerequisites from formal language theory which seem to be necessary. Actually, almost systematically, these people have invoked the "misterious" notion of a matrix grammar with appearance checking (long name, indeed), as one not at all very popular in computer science higher education curricula. Unfortunately, this is true, the "old" stuff of regulated rewriting, with the roots in sixties and flourishing in seventies and eighties, is not taught in current courses of formal language theory, although, as we hopefuly will prove in this paper, at least basic regulations on the context-free grammars, such as matrix grammars (with appearance checking), are not only easy to understand, but also very powerful, from many points of view, and also raise interesting new research questions. Using matrix grammars is rather useful in many cases, because of several reasons: they involve context-free grammars, which is the most widely investigated and applied class of Chomsky grammars (we include here the regular grammars), they have beautiful normal forms, and, maybe more important, there are very powerful results, proved several decades ago by means of intricate techniques; by invoking such results we can make use "for free" of the power of these techniques. A typical illustration of this assertion is the characterization of recursively enumerable languages by means of matrix grammars with appearance checking (using λ-rules) in the strong binary normal form (we hope that this long list of technical terms will not scare also the reader of this text...). The rather complex proof of this result, based on a similar result of Rosenkrantz, from 1969, about programmed grammars, has the same main idea as that used by Minsky, when proving the equivalence of Turing machines with register machines (with two registers). Starting from matrix grammars in the mentioned normal form (or from register machines, in the case when recognizing strings) is now much easier to prove that other generative mechanisms are equivalent with Turing machines; the necessary proof will be easy just because the hard work was already done when passing from Turing machines to matrix grammars.

In short, we warmly encourage the reader not to give up in front of a half page definition, of a notion whose name is composed of four or more terms, assuring him/her that the reward is rather significant.

As mentioned in the abstract, we recall here several notions and results (for the reader with a mathematical background). Some of them are basic notions and notations, others were already used in many places in membrane computing; that is why we have called them "basic" and marked with a **B**. Others are a little bit more sophisticated and have appeared only incidentally in papers dealing with P systems, so that they are considered "advanced" and indicated by an **A**. Then, we have included some notions and results (marked with **R**, from "related") which, according to our knowledge, were not yet used in membrane computing investigations, but they are good candidates for being used. Maybe, in this way, new ideas for membrane computing researches will arise. Of course, this classification should be understood just as an attempt to make easier the life of the reader which is not interested in too many details from formal language theory, but in a quick training for entering the membrane computing area.

For additional information in formal language theory (or automata theory, which is only briefly mentioned here), the reader is referred to the many monographs in this area, starting with [9], and finishing with the comprehensive [8]. Also, the prerequisites chapter of [6] is a good synthetic source of information, thought incomplete for our needs (and containing additional material, too, mainly about characterizations of recursively enumerable languages, which does not seem of direct interest for membrane computing). In what concerns the precise historical references for the notions and the results mentioned here, the reader is referred to the monographs cited at the end of the paper.

2 Basic Notions and Notations about Words

A. Basic notations. An *alphabet* is a finite nonempty set of abstract symbols. For an alphabet V we denote by V^* the set of all strings of symbols from V. The empty string is denoted by λ. Mathematically speaking, V^* is the free monoid generated by V under the operation of *concatenation.* (The unit element of this monoid is λ.) The set of nonempty strings over V, that is $V^* - \{\lambda\}$, is denoted by V^+. Each subset of V^* is called a *language* over V. A language which does not contain the empty string (hence being a subset of V^+) is said to be λ-*free.*

Example 1.

$V = \{a, b, c\}$ is an alphabet,

$x = aaabbbcaa = a^3b^3ca^2$ is a string over V,

$L = \{a^nb^n \mid n \geq 1\}$ is a language over V.

If $x = x_1x_2$, for some $x_1, x_2 \in V^*$, then x_1 is called a *prefix* of x and x_2 is called a *suffix* of x; if $x = x_1x_2x_3$ for some $x_1, x_2, x_3 \in V^*$, then x_2 is called a *substring* of x. The sets of all prefixes, suffixes, substrings of a string x are denoted by $Pref(x), Suf(x), Sub(x)$, respectively.

The *length* of a string $x \in V^*$ (the number of occurrences in x of symbols from V) is denoted by $|x|$. The number of occurrences of a given symbol $a \in V$ in $x \in V^*$ is denoted by $|x|_a$. If $x \in V^*$, $U \subseteq V$, then by $|x|_U$ we denote the length of the string obtained by erasing from x all symbols not in U, that is,

$$|x|_U = \sum_{a \in U} |x|_a.$$

For a language $L \subseteq V^*$, the set $length(L) = \{|x| \mid x \in L\}$ is called the *length set* of L.

Example 1. (continued)

$$|x| = 9,\ |x|_a = 5,$$
$$|x|_{\{a,c\}} = 6,$$
$$length(L) = \{2n \mid n \geq 1\}.$$

The set of symbols occurring in a string x is denoted by $alph(x)$. For a language $L \subseteq V^*$, we denote $alph(L) = \bigcup_{x \in L} alph(x)$. Observe that $alph(L)$ may be a proper subset of V.

Example 1. (continued)

$$alph(x) = \{a, b, c\},$$
$$alph(L) = \{a, b\}.$$

The *Parikh vector* associated with a string $x \in V^*$ with respect to the alphabet $V = \{a_1, \ldots, a_n\}$ is $\Psi_V(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$ (note that the ordering of the symbols from V is relevant). For $L \subseteq V^*$ we define $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$; this is called the *Parikh mapping* associated with V. If FL is a family of languages, then we denote by $PsFL$ the family of Parikh images of languages in FL.

Example 1. (continued)

$$\Psi_V(x) = (5, 3, 1),$$
$$\Psi_V(L) = \{(n, n, 0) \mid n \geq 1\}.$$

A set M of vectors in $\mathbf{N}^n$, for some $n \geq 1$, is said to be *linear* if there are the vectors $v_i \in \mathbf{N}^n$, $0 \leq i \leq m$, such that

$$M = \{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_1, \ldots, \alpha_m \in \mathbf{N}\}.$$

A finite union of linear sets is said to be *semilinear*.

A language $L \subseteq V^*$ is semilinear if $\Psi_V(L)$ is a semilinear set. The family of semilinear languages is denoted by $SLIN$.

Example 2.

$\{(n,n,0) \mid n \geq 1\}$ is a linear set,

$\{(n,n,0) \mid n \geq 1\} \cup \{(1,1,1)\}$ is a semilinear non-linear set,

$\{(nm) \mid n,m \geq 2\}$ is not semilinear,

$\{(n,m) \mid n \geq 1, 1 \leq m \leq 2^n\}$ is not semilinear,

Any infinite subset of $\mathbf{N}$ which does not contain an infinite arithmetical progression is not semilinear.

B. Operations with strings and languages. The boolean operations (with languages) are denoted as usual: $\cup$ – union, $\cap$ – intersection, C – complementation.

The *concatenation* of L_1, L_2 is $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$.

We define further:

$$\begin{aligned} &L^0 = \{\lambda\},\\ &L^{i+1} = LL^i,\ i \geq 0,\\ &L^* = \bigcup_{i=0}^{\infty} L^i \text{ (the } *\text{-Kleene closure)},\\ &L^+ = \bigcup_{i=1}^{\infty} L^i \text{ (the } +\text{-Kleene closure)}. \end{aligned}$$

A mapping $h : V \longrightarrow U^*$, extended to $h : V^* \longrightarrow U^*$ by $h(\lambda) = \{\lambda\}$ and $h(x_1x_2) = h(x_1)h(x_2)$, for $x_1, x_2 \in V^*$, is called a *morphism.* If $h(a) \neq \lambda$ for each $a \in V$, then h is a λ-*free* morphism.

A. A morphism $h : V^* \longrightarrow U^*$ is called a *coding* if $h(a) \in U$ for each $a \in V$ and a *weak coding* if $h(a) \in U \cup \{\lambda\}$ for each $a \in V$. If $h : (V_1 \cup V_2)^* \longrightarrow V_1^*$ is the morphism defined by $h(a) = a$ for $a \in V_1$, and $h(a) = \lambda$ otherwise, then we say that h is a *projection* (associated with V_1) and we denote it by pr_{V_1}. For a morphism $h : V^* \longrightarrow U^*$, we define a mapping $h^{-1} : U^* \longrightarrow 2^{V^*}$ (and we call it an *inverse morphism*) by $h^{-1}(w) = \{x \in V^* \mid h(x) = w\}$.

If $L \subseteq V^*, k \geq 1$, and $h : V^* \longrightarrow U^*$ is a morphism such that $h(x) \neq \lambda$ for each $x \in Sub(L), |x| = k$, then we say that h is k-*restricted* on L.

R. For $x, y \in V^*$ we define their *shuffle* by

$$\begin{aligned} x \sqcup\!\sqcup\, y = \{x_1y_1 \ldots x_ny_n \mid x = x_1 \ldots x_n, y = y_1 \ldots y_n,\\ x_i, y_i \in V^*, 1 \leq i \leq n, n \geq 1\}. \end{aligned}$$

B. In general, if we have an n-ary operation on strings, $g : V^* \times \ldots \times V^* \longrightarrow 2^{U^*}$, we extend it to languages over V by

$$g(L_1, \ldots, L_n) = \bigcup_{\substack{x_i \in L_i \\ 1 \leq i \leq n}} g(x_1, \ldots, x_n).$$

A family FL of languages is *closed* under an n-ary operation g if, for all languages $L_1, \dots, L_n$ in FL, the language $g(L_1, \dots, L_n)$ is also in FL.

A. The *left quotient* of a language $L_1 \subseteq V^*$ with respect to $L_2 \subseteq V^*$ is

$$L_2 \backslash L_1 = \{w \in V^* \mid \text{there is } x \in L_2 \text{ such that } xw \in L_1\}.$$

The *left derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is

$$\partial^l_x(L) = \{w \in V^* \mid xw \in L\}.$$

The *right quotient* and the *right derivative* are defined in a symmetric manner:

$$\begin{aligned} &L_1/L_2 = \{w \in V^* \mid \text{there is } x \in L_2 \text{ such that } wx \in L_1\}, \\ &\partial^r_x(L) = \{w \in V^* \mid wx \in L\}. \end{aligned}$$

A. A language that can be obtained from the letters of an alphabet V and λ by using finitely many times the operations of union, concatenation, and Kleene $*$ is called *regular*; also the empty language is said to be regular.

A family of languages is *nontrivial* if it contains at least one language different from $\emptyset$ and $\{\lambda\}$. (We use here the word "family" synonymously with "set" or "collection".)

From now on, all families of languages we consider are supposed to be nontrivial.

A family of languages is called a *trio* if it is closed under λ-free morphisms, inverse morphisms, and intersection with regular languages. A trio closed under union is called a *semi-AFL* (AFL = abstract family of languages). A semi-AFL closed under concatenation and Kleene $+$ is called an *AFL*. A trio/semi-AFL/AFL is said to be *full* if it is closed under arbitrary morphisms (and Kleene $*$ in the case of AFL's). A family of languages closed under none of the six AFL operations is called an *anti-AFL*.

R. Here are some basic results related to abstract families of languages:

1. The family of regular languages is the smallest full trio.
2. Each (full) semi-AFL closed under Kleene $+$ is a (full) AFL.
3. If FL is a family of λ-free languages which is closed under concatenation, λ-free morphisms, and inverse morphisms, then FL is closed under intersection with regular languages and union, hence FL is a semi-AFL. (If FL is also closed under Kleene $+$, then it is an AFL.)
4. If FL is a family of languages closed under intersection with regular languages, union with regular languages, and substitution with regular languages, then FL is closed under inverse morphisms.

5. Every semi-AFL is closed under substitution with λ-free regular languages. Every full semi-AFL is closed under substitution with arbitrary regular languages and under left and right quotients with regular languages.
6. A family of λ-free languages is an AFL if it is closed under concatenation, λ-free morphisms, inverse morphisms, and Kleene $+$.
7. A family of languages that is closed under intersection with regular languages, union with regular languages, substitution by λ-free regular languages, and restricted morphisms is closed also under inverse morphisms.

B. Chomsky grammars. Generally speaking, a *grammar* is a (finite) device *generating* in a well specified sense the strings of a language (hence defining a set of syntactically correct strings). The Chomsky grammars are particular cases of *rewriting systems*, where the operation used in processing the strings is the rewriting (the replacement of a "short" substring of the processed string by another short substring).

A *Chomsky grammar* is a quadruple $G = (N, T, S, P)$, where N, T are disjoint alphabets, $S \in N$, and P is a finite subset of $(N \cup T)^*N(N \cup T)^* \times (N \cup T)^*$.

The alphabet N is called the *nonterminal alphabet*, T is the *terminal alphabet*, S is the *axiom*, and P is the set of *production rules* of G. The rules (we also say *productions*) (u, v) of P are written in the form $u \to v$. Note that $|u|_N \geq 1$.

For $x, y \in (N \cup T)^*$ we write

$$x \Longrightarrow_G y \text{ iff } x = x_1ux_2, y = x_1vx_2, \\ \text{for some } x_1, x_2 \in (N \cup T)^* \text{ and } u \to v \in P.$$

One says that x *directly derives* y (with respect to G). When G is understood we write $\Longrightarrow$ instead of $\Longrightarrow_G$. Each string $w \in (N \cup T)^*$ such that $S \Longrightarrow_G^* w$ is called a *sentential form.*

The language generated by G, denoted by $L(G)$, is defined by

$$L(G) = \{x \in T^* \mid S \Longrightarrow^* x\}.$$

Two grammars G_1, G_2 are called *equivalent* if $L(G_1) - \{\lambda\} = L(G_2) - \{\lambda\}$ (the two languages coincide modulo the empty string).

If in $x \Longrightarrow y$ above we have $x = x_1ux_2$, with $x_1 \in T^*$, then the derivation step is *leftmost* and we write $x \Longrightarrow_{left} y$. The leftmost language generated by the grammar G is obtained by derivations where every step is leftmost and is denoted by $L_{left}(G)$.

Example 3. For the grammar

$$G_1 = (\{S\}, \{a, b\}, S, \{S \to aSb,\ S \to ab\})$$

we obviously obtain

$$L(G_1) = \{a^n b^n \mid n \geq 1\}.$$

Indeed, each derivation in G_1 consists of $m \geq 0$ applications of the rule $S \rightarrow aSb$, and this produces the string $a^m S b^m$, and ends by using the rule $S \rightarrow ab$, which leads to the terminal string $a^{m+1} b^{m+1}$. Thus, each string generated by G is of the form $a^n b^n$, for some $n \geq 1$; conversely, each string of this form can be generated by G_1.

Example 4. The grammar

$$G_2 = (\{S, B\}, \{a, b, c\}, S, \{S \rightarrow aSBc,\ S \rightarrow abc,\ cB \rightarrow Bc,\ bB \rightarrow bb\})$$

generates the language

$$L(G_2) = \{a^n b^n c^n \mid n \geq 1\}.$$

The proof of this assertion is left to the reader.

B. According to the form of their rules, the Chomsky grammars are classified as follows. A grammar $G = (N, T, S, P)$ is called:

- *length-increasing*, if for all $u \rightarrow v \in P$ we have $|u| \leq |v|$.
- *context-sensitive*, if each $u \rightarrow v \in P$ has $u = u_1 A u_2, v = u_1 x u_2$, for $u_1, u_2 \in (N \cup T)^*, A \in N$, and $x \in (N \cup T)^+$. (In length-increasing and context-sensitive grammars the production $S \rightarrow \lambda$ is allowed, providing that S does not appear in the right-hand members of rules in P.)
- *context-free*, if each production $u \rightarrow v \in P$ has $u \in N$.
- *linear*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T^* \cup T^* N T^*$.
- *right-linear*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T^* \cup T^* N$.
- *left-linear*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T^* \cup N T^*$.
- *regular*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

The arbitrary, length-increasing, context-free, and regular grammars are also said to be of *type* 0, *type* 1, *type* 2, and *type* 3, respectively.

Note that the grammar G_1 from Example 3 is linear, while the grammar G_2 from Example 4 is length-increasing.

The family of languages generated by length-increasing grammars is equal to the family of languages generated by context-sensitive grammars; the families of languages generated by right- or by left-linear grammars coincide and they are equal to the family of languages generated by regular grammars, as well as with the family of regular languages (as defined some pages above).

We denote by *RE, CS, CF, LIN,* and *REG* the families of languages generated by arbitrary, context-sensitive, context-free, linear, and regular grammars, respectively (RE stands for *recursively enumerable*). By FIN we denote the family of finite languages.

Therefore, for the grammars from Examples 3 and 4 we have $L(G_1) \in LIN$ and $L(G_2) \in CS$.

The following strict inclusions hold:

$$FIN \subset REG \subset LIN \subset CF \subset CS \subset RE.$$

This is *the Chomsky hierarchy*, the constant reference for investigations related to the power of membrane systems (and of any new types of computing devices). This important role of Chomsky hierarchy is due to several reasons: the family RE of languages generated by type-0 Chomsky grammars is exactly the family of languages which are recognized by Turing machines (see below), and according to Turing-Church thesis this is the maximal level of algorithmic computability; the Chomsky hierarchy is well structured, hence we have a detailed classification of computing machineries (in between the above mentioned classes of grammars there are other types of grammars which were considered in the literature); there are many results in this area, formal language theory is a well developed foundamental branch of theoretical computer science (some decades ago it was said that formal language theory is "the flower of theoretical computer science" – Aho, Salomaa, etc).

Table 1. Closure properties of the families in the Chomsky hierarchy

	RE	CS	CF	LIN	REG
Union	Y	Y	Y	Y	Y
Intersection	Y	Y	N	N	Y
Complement	N	Y	N	N	Y
Concatenation	Y	Y	Y	N	Y
Kleene $*$	Y	Y	Y	N	Y
Intersection with regular languages	Y	Y	Y	Y	Y
Morphisms	Y	N	Y	Y	Y
λ-free morphisms	Y	Y	Y	Y	Y
Inverse morphisms	Y	Y	Y	Y	Y
Left/right quotient	Y	N	N	N	Y
Left/right quotient with regular languages	Y	N	Y	Y	Y
Left/right derivative	Y	Y	Y	Y	Y
Shuffle	Y	Y	N	N	Y

We also point here a major feature of the formal grammar approach to computability, which can be considered a drawback from the point of view of practical computer science: the language generated by a grammar consists of all strings obtained at the end of "successful" (terminal) derivations; the "unsuccessful" derivations are ignored, hence we adopt a positivistic approach, which, because of the nondeterministic behavior of grammars (the rule to be used at a given step and the substring to be rewritten by that rule are nondeterministically chosen) is easy to handle at the mathematical level, but not at the practical level (for instance, when implementing/simulating a grammar on a computer).

A. The closure properties of the families listed above are indicated in Table 1 (Y stands for **yes** and N for **no**).

Therefore, *RE, CF, REG* are full AFL's, *CS* is an AFL (not full), and *LIN* is a full semi-AFL.

B. **Normal forms.** From many points of view, in particular, in proofs, it is useful to characterize a family of languages by grammars of the corresponding type but of precise particular forms, called normal forms. Such characterizations (normal forms) exist for all families in the Chomsky hierarchy, but we consider here only some of those referring to RE languages.

B. **Theorem 1.** (Kuroda normal form) *For every type-0 grammar G, an equivalent grammar $G' = (N, T, S, P)$ can be effectively constructed, with the rules in P of the forms $A \to BC, A \to a, A \to \lambda, AB \to CD$, for $A, B, C, D \in N$ and $a \in T$.*

A. **Theorem 2.** (Penttonen normal form) *For every type-0 grammar G, an equivalent grammar $G' = (N, T, S, P)$ can be effectively constructed, with the rules in P of the forms $A \to x$, $x \in (N \cup T)^*, |x| \leq 2$, and $AB \to AC$ with $A, B, C \in N$.*

Similar results hold true for length-increasing grammars; then rules of the form $A \to \lambda$ are no longer allowed, but only a completion rule $S \to \lambda$ if the generated language should contain the empty string.

R. **Theorem 3.** (Geffert normal forms) (1) *Each recursively enumerable language can be generated by a grammar $G = (N, T, S, P)$ with $N = \{S, A, B, C\}$ and the rules in P of the forms $S \to uSv, S \to x$, with $u, v, x \in (T \cup \{A, B, C\})^*$, and only one non-context-free rule, $ABC \to \lambda$.*

(2) *Each recursively enumerable language can be generated by a grammar $G = (N, T, S, P)$ with $N = \{S, A, B, C, D\}$ and the rules in P of the forms $S \to uSv$, $S \to x$, with $u, v, x \in (T \cup \{A, B, C, D\})^*$, and only two non-context-free rules, $AB \to \lambda, CD \to \lambda$.*

A linear grammar with only one nonterminal is said to be *minimal.* Thus, the Geffert normal forms say that each recursively enumerable language can be obtained from a minimal linear language by applying the reduction rule $ABC \to \lambda$, or the reduction rules $AB \to \lambda, CD \to \lambda$.

B. Necessary conditions. For a language $L \subseteq V^*$, we define the equivalence relation $\sim_L$ over V^* by $x \sim_L y$ iff $(uxv \in L \Leftrightarrow uyv \in L)$ for all $u, v \in V^*$. Then $V^*/\sim_L$ is called the *syntactic monoid* of L.

Theorem 4. (Myhill–Nerode theorem) *A language $L \subseteq V^*$ is regular iff $V^*/\sim_L$ is finite.*

Theorem 5. (Bar-Hillel/$uvwxy$/pumping lemma) *If $L \in CF, L \subseteq V^*$, then there are $p, q \in \mathbf{N}$ such that every $z \in L$ with $|z| > p$ can be written in the form $z = uvwxy$, with $u, v, w, x, y \in V^*$, $|vwx| \leq q, vx \neq \lambda$, and $uv^iwx^iy \in L$ for all $i \geq 0$.*

Theorem 6. (Parikh theorem) *Every context-free language is semilinear.* Otherwise written, we have the relation $PsCF \subset SLIN$.

Corollary 1. (i) *Every context-free language over a one-letter alphabet is regular.*

(ii) *The length set of a context-free language is a finite union of arithmetical progressions.*

The conditions of Theorems 4 – 6 are only necessary, not sufficient for a language to be in the corresponding family.

Using these necessary conditions (and related tools, not specified here) the following relations can be proved:

$$\begin{aligned}
L_1 &= \{a^nb^n \mid n \geq 1\} \in LIN - REG,\\
L_2 &= L_1L_1 \in CF - LIN,\\
L_3 &= \{a^nb^nc^n \mid n \geq 1\} \in CS - CF,\\
L_4 &= \{xcx \mid x \in \{a,b\}^*\} \in CS - CF,\\
L_5 &= \{a^{2^n} \mid n \geq 1\} \in CS - CF,\\
L_6 &= \{a^nb^mc^nd^m \mid n, m \geq 1\} \in CS - CF,\\
L_7 &= \{a^nb^m \mid n \geq 1, 1 \leq m \leq 2^n\} \in CS - CF,\\
L_8 &= \{a^nb^mc^p \mid 1 \leq n \leq m \leq p\} \in CS - CF,\\
L_9 &= \{x \in \{a,b\}^* \mid |x|_a = |x|_b\} \in CF - LIN,\\
L_{10} &= \{x \in \{a,b,c\}^* \mid |x|_a = |x|_b = |x|_c\} \in CS - CF.
\end{aligned}$$

R. The *Dyck language*, D_n, over $T_n = \{a_1, a_1', \ldots, a_n, a_n'\}$, $n \geq 1$, is the context-free language generated by the grammar

$$G = (\{S\}, T_n, S, \{S \to \lambda, S \to SS\} \cup \{S \to a_iSa_i' \mid 1 \leq i \leq n\}).$$

Intuitively, the pairs $(a_i, a_i'), 1 \leq i \leq n$, can be viewed as left and right parentheses, of different kinds. Then D_n consists of all strings of correctly nested parentheses.

Theorem 7. (Chomsky–Schützenberger theorem) *Every context-free language L can be written in the form $L = h(D_n \cap R)$, where h is a morphism, D_n, $n \geq 1$, is a Dyck language, and R is a regular language.*

B. Lindenmayer systems. Membrane systems are biologically inspired and, also, fundamentally parallel computing devices; these observations make their comparison with Lindenmayer systems rather natural.

A 0L (0-interactions Lindenmayer) system is a construct $G = (V, w, P)$, where V is an alphabet, $w \in V^*$ (axiom), and P is a finite set of rules of the form $a \to v$ with $a \in V, v \in V^*$, such that for each $a \in V$ there is at least one rule $a \to v$ in P (we say that P is *complete*). For $w_1, w_2 \in V^*$ we write $w_1 \Longrightarrow w_2$ if $w_1 = a_1 \ldots a_n, w_2 = v_1 \ldots v_n$, for $a_i \to v_i \in P, 1 \leq i \leq n$. The generated language is $L(G) = \{x \in V^* \mid w \Longrightarrow^* x\}$.

If for each rule $a \to v \in P$ we have $v \neq \lambda$, then we say that G is *propagating* (non-erasing); if for each $a \in V$ there is only one rule $a \to v$ in P, then G is said to be *deterministic.* If we distinguish a subset T of V and we define $L(G)$ as $L(G) = \{x \in T^* \mid w \Longrightarrow^* x\}$, then we say that G is *extended.* The family of languages generated by 0L systems is denoted by $0L$; we add the letters *P, D, E* in front of $0L$ if propagating, deterministic, or extended 0L systems are used, respectively.

A *tabled* 0L system, abbreviated T0L, is a system $G = (V, w, P_1, \ldots, P_n)$, such that each triple $(V, w, P_i), 1 \leq i \leq n$, is a 0L system; each P_i is called a *table*, $1 \leq i \leq n$. The generated language is defined by

$$L(G) = \{x \in V^* \mid w \Longrightarrow_{P_{j_1}} w_1 \Longrightarrow_{P_{j_2}} \ldots \Longrightarrow_{P_{j_m}} w_m = x,$$
$$m \geq 0, 1 \leq j_i \leq n, 1 \leq i \leq m\}.$$

(Each derivation step is performed by the rules of the same table.)

A T0L system is deterministic when each of its tables is deterministic. The propagating and the extended features are defined in the usual way.

The family of languages generated by T0L systems is denoted by $T0L$; the $ET0L, EDT0L$, etc. families are obtained in the same way as $E0L, ED0L$, etc.

Example 5. The D0L system

$$G = (\{a\}, a, \{a \to aa\})$$

generates the language $\{a^{2^n} \mid n \geq 1\}$, which is not context-free.

A. Example 6. The E0L system

$$G = (\{A, A', B, B', C, C', F, a, b, c\}, \{a, b, c\}, ABC, P),$$

with the rules

$$\begin{aligned}&A \to AA',\ A \to a,\ A' \to A',\ A' \to a,\ a \to F,\\ &B \to BB',\ B \to b,\ B' \to B',\ B' \to b,\ b \to F,\\ &C \to CC',\ C \to c,\ C' \to C',\ C' \to c,\ c \to F,\\ &F \to F,\end{aligned}$$

generates the non-context-free language $\{a^n b^n c^n \mid n \geq 1\}$ (the reader is asked to check this assertion).

B. The *D0L* family is incomparable with *FIN, REG, LIN, CF*, whereas $E0L$ strictly includes the family CF; $ET0L$ is the largest family of Lindenmayer languages with 0-interactions, it is strictly included in CS, and it is a full AFL.

Here are some languages which are not in given L families:

$$\begin{aligned}&L_{11} = \{a, aa\} \notin D0L,\\ &L_{12} = \{a^m b^n a^m \mid 1 \leq m \leq n\} \notin E0L,\\ &L_{13} = \{a^n \mid n \text{ is a prime number}\} \notin ET0L,\\ &L_{14} = \{w \in \{a,b\}^* \mid |w|_b = 2^{|w|_a}\} \notin ET0L.\end{aligned}$$

The languages L_{13}, L_{14} illustrate the strictness of the inclusion $PsET0L \subset PsCS$.

It is not known whether or not $PsE0L$ is strictly included into $PsET0L$.

R. An interesting feature of a D0L system, $G = (V, w, P)$, is that it generates its language in a *sequence*, $L(G) = \{w = w_0, w_1, w_2, \ldots\}$, such that $w_0 \Longrightarrow w_1 \Longrightarrow w_1 \Longrightarrow \ldots$. Thus, we can define the *growth function* of G, denoted by $growth_G : \mathbf{N} \longrightarrow \mathbf{N}$, by

$$growth_G(n) = |w_n|,\ n \geq 0.$$

R. **Descriptional complexity.** A given language can be generated by infinitely many different grammars, but it is natural to look for grammars which are as simple as possible from various points of view. To this aim we need measures of grammar complexity.

Having a class X of grammars, a descriptional complexity measure (we also say *measure of syntactical complexity*) is a mapping $K : X \longrightarrow \mathbf{N}$ which is extended to languages generated by elements of X by $K(L) = \min\{K(G) \mid L = L(G), G \in X\}$. If necessary, then we also write $K_X(L)$, to specify the class of grammars used.

Here are three basic measures for context-free languages (they can be easily extended to non-context-free grammars). For a context-free grammar $G =$

(N, T, S, P) we define

$$\begin{aligned}
&Var(G) = \text{card}(N),\\
&Prod(G) = \text{card}(P),\\
&Symb(G) = \sum_{r \in P} Symb(r), \text{ where } Symb(r : A \to x) = |x| + 2.
\end{aligned}$$

For $L \in CF$ and $M \in \{Var, Prod, Symb\}$ we define

$$M_{CF}(L) = \inf\{M(G) \mid L = L(G), G \text{ a context-free grammar}\}.$$

A complexity measure M is called *non-trivial* if for each n there is a grammar G_n such that $M(L(G_n)) > n$; M is said to be *connected* if there is n_0 such that for each $n \geq n_0$ there is G_n with $M(L(G_n)) = n$.

All measures $Var_{CF}, Prod_{CF}, Symb_{CF}$ are connected (even with respect to the family of regular languages). Two measures of syntactical complexity cannot generally be simultaneously improved: there are languages L such that if we find a grammar for L which is optimal from the point of view of one measure, then this grammar is not optimal from the point of view of the other measure.

Most decision problems with respect to measures $Var_{CF}, Prod_{CF}, Symb_{CF}$ are unsolvable (of course, with respect to context-free grammars): for arbitrary G and n, it is not decidable whether or not $K(L(G)) = n$; it is not possible to effectively compute $K(L(G))$, for arbitrary G; given a grammar G, it is not possible to construct algorithmically a grammar G' such that $K(L(G)) = K(G')$.

B. Automata and transducers. Automata are computing devices which start from the strings over a given alphabet and *analyze* them (we also say *recognize*), telling us whether or not the input string belongs to a specified language.

The five basic families of languages in the Chomsky hierarchy, *REG, LIN, CF, CS, RE*, are also characterized by recognizing automata. These automata are: the finite automaton, the one-turn pushdown automaton, the pushdown automaton, the linearly bounded automaton, and the Turing machine, respectively. We present here only two of these devices, those which, in some sense, define the two poles of computability: finite automata and Turing machines.

A (nondeterministic) *finite automaton* is a construct

$$M = (K, V, s_0, F, \delta),$$

where K and V are disjoint alphabets, $s_0 \in K, F \subseteq K$, and $\delta : K \times V \longrightarrow 2^K$; K is the set of states, V is the alphabet of the automaton, s_0 is the initial state, F is the set of final states, and δ is the transition mapping. If $card(\delta(s, a)) \leq 1$ for all $s \in K, a \in V$, then we say that the automaton is *deterministic*. A relation $\vdash$ is defined in the following way on the set $K \times V^*$:

for $s, s' \in K, a \in V, x \in V^*$, we write $(s, ax) \vdash (s', x)$ if $s' \in \delta(s, a)$; by definition, $(s, \lambda) \vdash (s, \lambda)$. If $\vdash^*$ is the reflexive and transitive closure of the relation $\vdash$, then the language of the strings recognized by automaton M is defined by

$$L(M) = \{x \in V^* \mid (s_0, x) \vdash^* (s, \lambda), s \in F\}.$$

It is known that both deterministic and nondeterministic finite automata characterize the same family of languages, namely REG. The power of finite automata is not increased if we also allow λ-*transitions*, that is, if δ is defined on $K \times (V \cup \{\lambda\})$ (the automaton can also change its state when reading no symbol on its tape), or when the input string is scanned in a two-way manner, going along it to right or to left, without changing its symbols.

R. An important related notion is that of a *sequential transducer* which is nothing else than a finite automaton with outputs associated with its moves; we do not enter here into details and refer the reader to the general formal language theory literature.

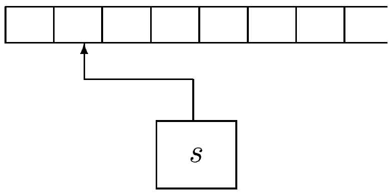

Figure 1: A finite automaton

B. We can imagine a finite automaton as in Figure 1, where we distinguish the input tape, on whose cells we write the symbols of the input alphabet, the read head, which scans the tape from the left to the right, and the memory, able to hold a state from a finite set of states. In the same way, a sequential transducer is a device as in Figure 2, where we also have an output tape, where the write head can write the string obtained by translating the input string.

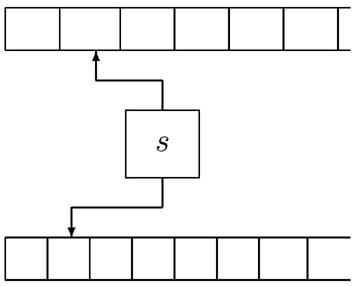

Figure 2: A sequential transducer

A. A *Turing machine* is a construct

$$M = (K, V, T, B, s_0, F, \delta),$$

where K, V are disjoint alphabets (the set of states and the tape alphabet), $T \subseteq V$ (the input alphabet), $B \in V - T$ (the blank symbol), $s_0 \in K$ (the initial state), $F \subseteq K$ (the set of final states), and δ is a partial mapping from $K \times V$ to the power set of $K \times V \times \{L, R\}$ (the move mapping; if $(s', b, d) \in \delta(s, a)$, for $s, s' \in K, a, b \in V$, and $d \in \{L, R\}$, then the machine reads the symbol a in state s and passes to state s', replaces a with b, and moves the read-write head to the left when $d = L$ and to the right when $d = R$). If $card(\delta(s, a)) \leq 1$ for all $s \in K, a \in V$, then M is said to be *deterministic*.

An *instantaneous description* of a Turing machine as above is a string xsy, where $x \in V^*, y \in V^*(V - \{B\}) \cup \{\lambda\}$, and $s \in K$. In this way we identify the contents of the tape, the state, and the position of the read-write head: it scans the first symbol of y. Observe that the blank symbol may appear in x, y, but not in the last position of y; both x and y may be empty. We denote by ID_M the set of all instantaneous descriptions of M.

On the set ID_M one defines the *direct transition* relation $\vdash_M$ as follows:

$$\begin{aligned}
&xsay \vdash_M xbs'y \text{ iff } (s', b, R) \in \delta(s, a),\\
&xs \vdash_M xbs' \text{ iff } (s', b, R) \in \delta(s, B),\\
&xcsay \vdash_M xs'cby \text{ iff } (s', b, L) \in \delta(s, a),\\
&xcs \vdash_M xs'cb \text{ iff } (s', b, L) \in \delta(s, B),
\end{aligned}$$

where $x, y \in V^*, a, b, c \in V, s, s' \in K$.

The language recognized by a Turing machine M is defined by

$$L(M) = \{w \in T^* \mid s_0 w \vdash_M^* xsy \text{ for some } s \in F, x, y \in V^*\}.$$

(This is the set of all strings such that the machine reaches a final state when starting to work in the initial state, scanning the first symbol of the input string.)

It is also customary to define the language accepted by a Turing machine as consisting of the input strings $w \in T^*$ such that the machine, starting from the configuration $s_0 w$, reaches a configuration where no further move is possible (we say that the machine *halts*); in this case, the set F of final states is no longer necessary. The two modes of defining the language $L(M)$ are equivalent, the identified families of languages are the same, namely RE, and this is true both for deterministic and nondeterministic machines.

Graphically, a Turing machine can be represented as a finite automaton (Figure 1). The difference between a finite automaton and a Turing machine is visible only in their functioning: the Turing machine can move its head in both directions and it can rewrite the scanned symbol, possibly erasing it (replacing it with the blank symbol).

A Turing machine can be also viewed as a mapping-defining device, not only as a mechanism defining a language, but we do not enter here into details (roughly speaking, the computed mapping links two instantaneous descriptions of the tape, an input one and a halting one which gives the value of the function).

When working on an input string a Turing machine is allowed to use as much tape as it needs. Note that finite automata use (in the read manner) only the cells where the input string is written. A Turing machine allowed to use only a working space linearly bounded with respect to the length of the input string is called a *linearly bounded automaton.* These machines characterize the family CS.

R. Register machines. A very powerful tehnique in automata and language theory is to consider the strings over an alphabet with k symbols as numbers in base $k+1$ (none of the symbols stands for zero), to work with numbers, and eventually to return to strings. This is the basic idea of register machines, already investigated in sixties (and it is also the proof idea of a central result in regulated rewriting, namely, of the characterization of recursively enumerable languages by context-free matrix grammars with appearance checking – see definitions in the corresponding section below).

We consider here the register machines in the form used in [4]:

An *n-register machine* is a construct $M = (n, R, i_0, f)$, where: (a) n is a natural number (the number of registers the machine may use); (b) R is a set of *labeled program instructions* of the form $k : (op(i), l, m)$ such that $op(i)$ is an operation on register i of M and k, l, m are labels from a set of labels $Lab(M)$ ($Lab(M)$ labels the program instructions of M in a one-to-one manner), $k \neq f$, l is the label for continuing the program if $op(i)$ can be applied to register i and m is the label for continuing the program if it is not possible to apply $op(i)$ to register i; (c) f is the *final label*, to which we assign the instruction *end*, which halts the program of the register machine M; (d) i_0 is the *initial label* to start the program.

We will use the following program instructions $(op(i), l, m)$:

- $(S(i), l, m)$: if possible (i.e., if the contents of register i is greater than zero), subtract 1 from register i and go to label l, otherwise skip, i.e., do not change the contents of register i, and continue with the instruction with label m;
- $(A(i), h, h)$: add 1 to register i and continue the program with the instruction at label h; obviously, the operation $A(i)$ is always possible, hence both labels where to continue have to be the same.

In some variants of register machines, h as well as one out of l, m have to be $k+1$, with k being the label of the program instruction under consideration; we do not consider such restrictions (hence our variant is more powerful), because writing programs for a register machine becomes easier in the variant adopted here.

An n-register machine M can be used to compute a (partially defined) function $g : \mathbf{N} \longrightarrow \mathbf{N}$ in the following way: M starts with $m \in \mathbf{N}$ in register 1; if M halts in the final label f and with the contents of register 1 being r, then we say that M has computed $g(m) = r$, otherwise (that is, if M does not halt in the final label f when started with m in register 1), $g(m)$ remains undefined.

Register machines (with two registers) are equivalent with Turing machines, in the sense that they compute the same family of (partially defined) functions.

Example 7. We consider a 2-register machine which computes the partial function $g : \mathbf{N} \longrightarrow \mathbf{N}$ which yields m for even numbers $2m$ and remains undefined for odd natural numbers.

Such a machine is $M = (2, R, 0, 5)$, with the following instructions (0 is the initial label):

$0 : (S(1), 1, 3)$,

$1 : (A(2), 2, 2)$,

$2 : (S(1), 0, 2)$,

$3 : (S(2), 4, 5)$,

$4 : (A(1), 3, 3)$,

$5 :$ *end*.

This "program" works as follows: If at label 0 the contents of register 1 is zero, then we jump to label 3, where in a loop which repeatedly uses instructions 3 and 4, M copies register 2 (which at the beginning of the loop contains half of the initial value from register 1) back into register 1, and, after this copying, M stops in label 5; otherwise, if at label 0 the subtraction is possible, we subtract 1 and continue at label 1, where we add 1 to register 2; then again we try to subtract 1 at label 2; if this is not possible, then we enter an infinite loop in 2; otherwise we subtract 1 and return to label 0. It is now obvious to see that in fact M computes the partial function $g : \mathbf{N} \longrightarrow \mathbf{N}$ which computes $m/2$ for an even number m and remains undefined for an odd natural number m.

B. Regulated rewriting. The context-free grammars are not powerful enough for covering most of the important syntactic constructions in natural and artificial languages, while the context-sensitive grammars are too powerful (for instance, the family CS has many negative decidability properties and the derivations in a non-context-free grammar cannot be described by a tree). A way to overpass this difficulty is to restrict the freedom of using the rules of a context-free grammar, and this has motivated the introduction of several dozens of controls/regulations of the derivation in context-free grammars.

The oldest one, the *matrix grammars*, were introduced, in a particular form, already in 1965, and it turns out to be very useful for membrane computing. We present it in a gradual (historical) manner, first in the particular and weaker form "without appearance checking", and then in the general form, the one which is equal in power to Chomsky type-0 grammars.

A context-free *matrix* grammar (without appearance checking) is a construct $G = (N, T, S, M)$, where N, T are disjoint alphabets (of nonterminals and terminals, respectively), $S \in N$ (axiom), and M is a finite set of *matrices*, that is, sequences of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$. For a string x, a matrix $m = (r_1, \ldots, r_n)$ is executed by applying productions $r_1, \ldots, r_n$ one after the other, following the order they are listed in. Formally, we write $y \Longrightarrow_m z$ if there is a matrix $m = (A_1 \to x_1, \ldots, A_n \to x_n)$ in M and the strings $w_1, w_2, \ldots, w_{n+1}$ in $(N \cup T)^*$ such that $y = w_1, w_{n+1} = z$ and for each $i = 1, 2, \ldots, n$ we have $w_i = w'_i A_i w''_i, w_{i+1} = w'_i x_i w''_i$. If the matrix m is understood, then we write $\Longrightarrow$ instead of $\Longrightarrow_m$. The reflexive and transitive closure of this relation is denoted by $\Longrightarrow^*$. Then, the generated language is

$$L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}.$$

The family of languages generated by context-free matrix grammars is denoted by MAT^λ (the superscript indicates that λ-rules are allowed); when using only λ-free rules, we denote the corresponding family by MAT.

Example 8. The non-context-free language $L_3 = \{a^n b^n c^n \mid n \geq 1\}$ mentioned above can be generated by the matrix grammar

$$\begin{aligned}
G &= (\{S, A, B, C\}, \{a, b, c\}, S, M), \text{ with the matrices}\\
M &= \{(S \to ABC),\\
&\quad (A \to aA, B \to bB, C \to cC),\\
&\quad (A \to a, B \to b, C \to c)\}.
\end{aligned}$$

The reader can easily check this assertion: the use of each matrix increases synchronously the number of occurrences of symbols a, b, c.

It is also easy to construct matrix grammars for the non-context-free languages L_4, L_6 considered above. This might not be similarly easy also for language L_7, hence we give a matrix grammar for this language.

Example 9. Consider the following matrix grammar:

$$\begin{aligned}
G \; = \; & (\{S, A, B, C, D, E\}, \{a, b\}, S, M), \text{ with } M \text{ containing the matrices}\\
& m_1 = (S \to AE),\\
& m_2 = (A \to A, E \to DD),\\
& m_3 = (A \to aB),\\
& m_4 = (B \to B, D \to EE),\\
& m_5 = (B \to aA),
\end{aligned}$$

$$\begin{aligned}
&m_6 = (A \to C),\\
&m_7 = (B \to C),\\
&m_8 = (C \to a),\\
&m_9 = (C \to C, E \to b),\\
&m_{10} = (C \to C, D \to b),\\
&m_{11} = (S \to ab).
\end{aligned}$$

Excepting the derivation $S \to ab$ (using the matrix m_{11}), all derivations in G begin by using the matrix m_1, and contain two main phases: one which uses the matrices m_8, m_9, m_{10} (in the presence of C), and an initial one which uses the matrices m_2, m_3, m_4, m_5. These phases are separated by a step which uses either m_6 or m_7. In the first phase, the matrices m_2 and m_4 can double the number of occurrences of the symbols D and E. This is possible only in the presence of symbols A and B, respectively. The change from A to B and conversely introduces an occurrence of the terminal a. Since it is not necessary to double all occurrences of D and E by matrices m_2 and m_4, the number of occurrences of D and E is smaller than or equal to 2^n, where n is the number of occurrences of the symbol a. The second phase of the derivation replaces each occurrence of D and E by b. Therefore, the generated string belongs to the language L_7.

Conversely, each string in L_7 can be generated by the grammar G, hence we have the equality $L(G) = L_7$.

We leave to the reader the task to write a grammar for the language L_8, and we only produce a grammar for the language L_{10}. Actually, because L_{10} is the permutation closure of the language L_3, we only indicate the way of constructing a grammar $G' = (N', T, S, M')$ which generates the permutation of the language generated by a given matrix grammar $G = (N, T, S, M)$. The idea is rather simple: For each terminal symbol a of G we introduce the symbol a' which is a nonterminal for G'. Then, G' contains all matrices of G, as well as the matrices

$$\begin{aligned}
&(a' \to b', b' \to a'),\ a, b \in T,\\
&(a' \to a),\ a \in T.
\end{aligned}$$

(Clearly, $N' = N \cup \{a' \mid a \in T\}$.) The matrices from $M' - M$ can be used at any time, and they permute the (primed) terminal symbols of G in an arbitrary manner, hence producing all permutations of strings in $L(G)$.

In this way we have obtained a proof for the important observation that the families MAT, MAT^λ are closed under permutation.

The following results about matrix languages are known:

1. $CF \subset MAT \subseteq MAT^\lambda \subset RE$.
2. $MAT \subset CS,\ CS - MAT^\lambda \neq \emptyset$.

3. Each language $L \in MAT^{\lambda}, L \subseteq a^*$, is regular. (This implies that the above language L_5 is not in MAT^{λ}.)

The last assertion has some interesting consequences:

1. $D0L - MAT^{\lambda} \neq \emptyset$, $PsD0L - PsMAT^{\lambda} \neq \emptyset$,
2. $PsCS - PsMAT^{\lambda} \neq \emptyset$.

The questions whether or not the inclusion $MAT \subseteq MAT^{\lambda}$ is proper and whether or not MAT^{λ} contains languages which are not context-sensitive are old *open problems* of this area.

B. In a matrix grammar (without appearance checking), when using a matrix, *all* rules are used, in the ordering imposed by the matrix. A powerful extension is provided by the possibility of skipping certain rules, and this leads to the definition of matrix grammars *with appearance checking.*

Such a grammar is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \rightarrow x_1, \ldots, A_n \rightarrow x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases; thus, (N, T, S, M) is a matrix grammar as above), and F is a set of occurrences of rules in M (N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, while the elements of M are called matrices).

For $y, z \in (N \cup T)^*$ we write $y \Longrightarrow z$ if there is a matrix $(A_1 \rightarrow x_1, \ldots, A_n \rightarrow x_n)$ in M and the strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $y = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either (1) $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$, or (2) $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \rightarrow x_i$ appears in F.

Therefore, the difference between a matrix grammar with appearance checking and one without appearance checking is the fact that in the former case we have at our disposal the set F, of *occurrences* of rules in the matrices of M (that is, if the same rule, say $A \rightarrow x$, appears several times in the matrices, only some of them can be present in F; we may interpret them as "marked", having some "flags" which distinguish them from the other rules); the rules of a matrix are applied in order, as usual, possibly skipping the rules in F if they cannot be applied. Thus, if a rule not in F is met, then it has to be used. If a rule from F is met, then we have two cases: if it can be applied, then it must be applied; if it cannot be applied (the nonterminal from its left hand member is not present in the current string), then the rule may be skipped. That is why the rules from F are said to be applied in the *appearance checking* mode. The information given by such an use of a rule from F is rather powerful, because in the case of skipping the rule we get a "negative information": a certain symbol *is not* present in the string. The example given below will illustrate both the definition of the derivation in a matrix grammar with appearance checking and the power of using the rules from the set F.

The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}^{λ} when using λ-rules and by MAT_{ac} when only λ-free rules are allowed.

It is known that

1. $MAT \subset MAT_{ac} \subset CS$,
2. $MAT^{\lambda} \subset MAT_{ac}^{\lambda} = RE$.

We have mentioned above that the language $L_5 = \{a^{2^n} \mid n \geq 1\}$ is not in the family MAT^{λ}. In contrast, this language can be generated by a matrix grammar using the appearance checking feature. Actually, we will produce a λ-free matrix grammar with appearance checking for the language $L_5' = \{ba^{2^n} \mid n \geq 1\}$. The initial symbol b can be erased by a restricted morphism or by a left derivative; because the family MAT_{ac} is closed under these operations, we obtain the fact that $L_5 \in MAT_{ac}$, hence a grammar for L_5 can also be obtained – but it will be a little bit more complex and we want to have a clearer example here.

Example 10. Consider the grammar

$$G = (\{S, A, B, X, Y, Z, \#\}, \{a, b\}, S, M, F),$$

with the following matrices:

$$\begin{aligned}
m_1 &= (S \to XA),\\
m_2 &= (X \to X, A \to BB),\\
m_3 &= (X \to Y, A \to \#),\\
m_4 &= (Y \to Y, B \to A),\\
m_5 &= (Y \to X, B \to \#),\\
m_6 &= (Y \to Z, B \to \#),\\
m_7 &= (Z \to Z, A \to a),\\
m_8 &= (Z \to b, A \to a).
\end{aligned}$$

The set F consists of all the rules having the symbol # in their right hand member.

Let us examine the work of this grammar. First, let us observe that # is a trap-symbol, once introduced, it cannot be removed, hence the sentential form will not lead to a terminal string. That is, the rules from F should be always "applied" by skipping them, hence when the sentential form does not contain the corresponding symbols A, B from the rules from the second positions of matrices m_3, m_5, m_6. Second, one sees that the symbols X, Y, Z are a sort of control symbols, the "real work" is done by the rules from the second place of matrices, under the control of the rules from the first position. Specifically, in the presence of X one doubles the length of the string (passing from a string of the form XA^m to a string of the form XB^{2m}), in the presence of Y one returns to a string of the form XA^p, while in the presence of Z we can pass to a terminal string.

We start by using m_1, hence producing the string XA. Assume that we have a sentential form XA^m, for some $m \geq 1$. As we have mentioned above, if we use m_3 in order to rewrite a string which contains at least one occurrence of A, then the trap-symbol # is introduced and the derivation will not lead to a terminal string. Thus, we have to use m_2 as much as possible, that is, until producing the string XB^{2m} (each use of the matrix replaces one occurrence of A by two occurrences of B). To such a string we can apply (only) m_3, and we obtain the string YB^{2m}. Observe the important contribution of the rule $A \rightarrow \#$: if we do not introduce the symbol #, then we are sure that *all* symbols A were rewritten, the doubling was complete. Now, matrix m_4 should be used as long as at least one occurrence of B is present: if at least one B is present, then the matrices m_5, m_6 will introduce the trap-symbol # . When we get the string YA^{2m} we can use either the matrix m_5, or the matrix m_6. In the former case we get the string XA^{2m}, hence the process can be repeated. In the latter case we get the string ZA^{2m}. From now on, only matrices m_7, m_8 can be used. Of course, m_8 should be used when only one occurrence of A is present, otherwise the string cannot be processes any more, in spite of the fact that it is a nonterminal one. Thus, we can repeatedly double the number of occurrences of A, and then we can turn to a terminal string. This means that the language L_5' is generated, indeed.

If matrix m_8 is replaced by $m_8' = (Z \rightarrow a, A \rightarrow a)$, then we generate the one-letter non-regular language $\{a^{2^n+1} \mid n \geq 1\}$.

B. The previous grammar had matrices of a rather restricted form. Actually, for each language in MAT_{ac}^{λ} (hence for each recursively enumerable language) one can find a grammar of this form. Specifically, the following important result holds.

A matrix grammar $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$, with these three sets mutually disjoint, and the matrices in M are in one of the following forms:

1. $(S \rightarrow XA)$, with $X \in N_1, A \in N_2$,
2. $(X \rightarrow Y, A \rightarrow x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$,
3. $(X \rightarrow Y, A \rightarrow \#)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \rightarrow \lambda, A \rightarrow x)$, with $X \in N_1, A \in N_2$, and $x \in T^*$.

Moreover, there is only one matrix of type 1 and F consists exactly of all rules $A \rightarrow \#$ appearing in matrices of type 3; # is a trap-symbol, once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

Theorem 8. *For each matrix grammar there is an equivalent matrix grammar in the binary normal form.*

For an arbitrary matrix grammar $G = (N, T, S, M, F)$, let us denote by $ac(G)$ the cardinality of the set $\{A \in N \mid A \to \alpha \in F\}$ and by $Var(G)$ the cardinality of N. From the construction in the proof of Theorem 8 (Lemma 1.3.7 in [3]) one can see that if we start from a matrix grammar G and we get the equivalent grammar G' in the binary normal form, then $ac(G') = ac(G)$. Moreover, it is known already from eighties that for each language $L \in RE$ there is a matrix grammar with appearance checking G such that $L(G) = L$ and $Var(G) \leq 6$.

This last result was recently improved in [4], where it was proved that four nonterminals are sufficient in order to characterize RE by matrix grammars and out of them only three are used in appearance checking rules. Of interest for membrane computing proved to be another result from [4]: if the total number of nonterminals is not restricted, then each recursively enumerable language can be generated by a matrix grammar G such that $ac(G) \leq 2$.

Consequently, to the properties of a grammar G in the binary normal form we can add the fact that $ac(G) \leq 2$. One says that this is *the strong binary normal form* for matrix grammars.

It is an *open problem* whether or not the results from [4] can be improved. In particular, it is of interest to find whether or not grammars G such that $ac(G) \leq 1$ can characterize RE (it was conjectured that the answer is negative, namely, that $ac(G) \leq 1$ implies $L(G) \in MAT^{\lambda}$).

R. As we have mentioned, there are many ways to control the derivation of a context-free grammar in such a way to generate non-context-free languages. Details can be found in [3]. We only recall here some basic definitions.

A context-free *programmed* grammar is a construct $G = (N, T, S, P)$, where N, T, S are as above, the set of nonterminals, the set of terminals, and the start symbol, and P is a finite set of productions of the form $(b : A \to z, E, F)$, where b is a label, $A \to z$ is a context-free production over $N \cup T$, and E, F are two sets of labels of productions of G. (E is said to be the *success field*, and F is the *failure field* of the production.) A production of G is applied as follows: if the context-free rule $A \to z$ can be successfully executed, then it is applied and the next production to be executed is chosen from those with the label in E, otherwise, we choose a production labeled by some element of F, and try to apply it. This type of programmed grammars is said to be with *appearance checking*; if no failure field is given for any of the productions, then a programmed grammar without appearance checking is obtained.

Sometimes it is useful to write a programmed grammar in the form $G = (N, T, S, P, \sigma, \varphi)$, where N, T, S are as above, P is a set of usual context-free rules and σ, φ are mappings from P to the power set of P; $\sigma(p), p \in P$, is the success field of the rule p (a rule in $\sigma(p)$ must be used after successfully applying the rule p), and $\varphi(p), p \in P$, is the failure field (a rule from $\varphi(p)$ must be considered when p cannot be applied). This way of writing a programmed grammar emphasizes the important feature of these devices: the control is imposed on the next-rule-to-be-used, on the sequence of rules, from a step to the next one (not by prescribing blocks of rules, as in a matrix grammar).

The family of languages generated by programmed grammar with arbitrary context-free rules is denoted by PR_{ac}^{λ}; when only λ-free rules are used one removes the superscript λ; when all failure fields are empty, then one removes the subscript *ac*.

One knows that $MAT_{\beta}^{\alpha} = PR_{\beta}^{\alpha}$ for all combinations of α and β (of course, α can be λ or can be missing, while β can be *ac* or can be missing).

A. A context-free *ordered* grammar is a system $G = (N, T, S, P, >)$, where N, T, S are as above, P is a finite set of context-free productions, and $>$ is a partial order relation over P. A production p can be applied to a sentential form x only if it can be applied as a context-free rule and there is no production $r \in P$ such that r is applicable and $r > p$ holds.

The family of languages generated by ordered grammars is denoted by ORD when only λ-free rules are used, and by ORD^{λ} when arbitrary context-free rules are allowed. The following relations are known:

1. $ET0L \subset ORD \subset MAT_{ac}$,
2. $ORD^{\lambda} \subset RE$.

A. Regulated applications of productions can also be based on checking the contents of the string to which a rule is to be used. We present here only the *random context grammars*, which are constructs $G = (N, T, S, P)$, where N, T, S are as above and P is a finite set of triples of the form $p = (A \to w; E, F)$, where $A \to w$ is a context-free production over $N \cup T$ and E, F are subsets of N. Then, p can be applied to a string $x \in (N \cup T)^*$ only if A appears in x, $E \subseteq alph(x) - \{A\}$, and $F \cap (alph(x) - \{A\}) = \emptyset$. If E or F is the empty set, then no condition is imposed by E or F, respectively. E is said to be the set of *permitting* and F is said to be the set of *forbidding* context conditions of p.

The elements of E, F can also be strings, and then all the strings from E should appear in the sentential form to be rewritten and no string from F should appear in this sentential form. Such grammars (called *semi-conditional*) generate all recursively enumerable or all context-sensitive languages, depending on whether λ-rules are used or not, respectively.

For random context grammars, we have the following results. Denote by RC_{ac}^{λ} the family of languages generated by random context grammars with erasing rules; when all sets F are empty, then the subscript *ac* is removed, and when only λ-free rules are used we remove the superscript λ. We have:

1. $CF \subset RC \subseteq MAT \subset RC_{ac} = MAT_{ac}$,

2. $RC \subseteq RC^\lambda \subseteq MAT^\lambda \subset RC^\lambda_{ac} = RE$.

R. Grammar systems. Another very fruitful idea for increasing the power of context-free grammars (in certain cases, also of regular grammars) is to consider distributed generative devices: constructs composed of several grammars working together according to a well-specified cooperation protocol. This leads to the idea of a *grammar system.* Two main classes of grammar systems have been investigated, the sequential ones (known under the name of *cooperating distributed grammar systems*) and the *parallel communicating grammar systems.* We briefly introduce here both these classes; details – including bibliographical references – can be found in [1] and [8].

R. A *cooperating distributed* (in short, CD) grammar system of degree $n, n \geq 1$, is a construct

$$\Gamma = (N, T, S, P_1, P_2, \dots, P_n),$$

where N, T are disjoint alphabets (the nonterminal and the terminal alphabet, respectively), $S \in N$ (the axiom), and $P_1, \dots, P_n$ are finite sets of context-free rules over $N \cup T$. For two sentential forms w, w' over $N \cup T$, we write $w \Longrightarrow_i^* w', w \Longrightarrow_i^{=k} w', w \Longrightarrow_i^{\leq k} w', w \Longrightarrow_i^{\geq k} w', w \Longrightarrow_i^t w'$, for some $k \geq 1$, if w' can be obtained from w (1) by any number of derivation steps, (2) by k derivation steps, (3) by at most k derivation steps, (4) by at least k derivation steps, (5) by a maximal number of derivation steps, using rules from P_i. (In the case of the t mode, no further derivation step $w' \Longrightarrow w''$ is possible by means of rules from P_i.) The language $L_\alpha(\Gamma)$, generated by Γ in the mode $\alpha \in \{*, t\} \cup \{\leq k, = k, \geq k \mid k \geq 1\}$, consists of all strings $x \in T^*$ such that $S \Longrightarrow_{j_1}^\alpha w_1 \Longrightarrow_{j_2}^\alpha \dots \Longrightarrow_{j_m}^\alpha w_m = x$, for some $m \geq 1, 1 \leq j_s \leq n, 1 \leq s \leq m$. (That is, the "components" $P_1, \dots, P_n$ of the system work in turn, according to the cooperation protocol α, on a common sentential form. When a terminal string is obtained it is said to be generated by the system.)

Example 11. For the CD grammar system

$$\begin{aligned} \Gamma \;=\; & (\{S, S', A, A', B, B'\}, \{a, b, c\}, S, P_1, P_2), \text{ with,} \\ & P_1 = \{S \to S', S' \to A'B', A \to a, B \to bc, A \to aA', B \to bB'c\}, \\ & P_2 = \{A' \to A, B' \to B\}, \end{aligned}$$

we obtain

$$\begin{aligned} L_{=2}(\Gamma) &= L_{\geq 2}(\Gamma) = \{a^n b^n c^n \mid n \geq 1\}, \\ L_{=k}(\Gamma) &= L_{\geq k}(\Gamma) = \emptyset, \text{ for all } k \geq 3. \end{aligned}$$

The reader can easily check these equalities (in the modes $= 2$ and ≥ 2 the system works like a matrix grammar, synchronously increasing the number of occurrences of each terminal symbol).

We denote by $CD_n(\alpha)$ the family of languages generated by CD grammar systems of degree at most $n \geq 1$, in the mode α; when the degree is not bounded the subscript n is replaced by $*$. We recall here only a few results about the power of CD grammar systems; further results can be found in [1] and in the corresponding chapter from [8].

1. $CF = CD_*(*) = CD_1(\alpha) = CD_*(=1) = CD_*(\leq k) = CD_*(\geq 1)$, for all $k \geq 1, \alpha \in \{t\} \cup \{=j, \geq j \mid j \geq 1\}$,
2. $CF = CD_1(t) = CD_2(t) \subset CD_3(t) = ET0L$,
3. $CD_*(\alpha) \subseteq MAT^\lambda$, for all $\alpha \in \{=k, \geq k \mid k \geq 1\}$,
4. $CD_n(=k) \subseteq CD_n(=rk)$, for all $n, r, k \geq 1$.

R. A *parallel communicating* (PC, for short) *grammar system* of degree n, $n \geq 1$, is a construct

$$\Gamma = (N, T, K, (S_1, P_1), \ldots, (S_n, P_n)),$$

where N, T, K are pairwise disjoint alphabets, with $K = \{Q_1, \ldots, Q_n\}$, $S_i \in N$, and P_i are finite sets of rewriting rules over $N \cup T \cup K, 1 \leq i \leq n$; the elements of N are *nonterminal* symbols, those of T are *terminals*; the elements of K are called *query symbols*; the pairs (S_i, P_i) are the *components* of the system. Note that the query symbols are associated in a one-to-one manner with the components. When discussing the type of the components in the Chomsky hierarchy, the query symbols are interpreted as nonterminals.

For $(x_1, \ldots, x_n), (y_1, \ldots, y_n)$, with $x_i, y_i \in (N \cup T \cup K)^*, 1 \leq i \leq n$ (we call such an n-tuple a *configuration*), and $x_1 \notin T^*$, we write $(x_1, \ldots, x_n) \Longrightarrow_r (y_1, \ldots, y_n)$ if one of the following two cases holds:

(i) $|x_i|_K = 0$ for all $1 \leq i \leq n$; then $x_i \Longrightarrow_{P_i} y_i$ or $x_i = y_i \in T^*, 1 \leq i \leq n$;

(ii) there is $i, 1 \leq i \leq n$, such that $|x_i|_K > 0$; we write such a string x_i as

$$x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_t Q_{i_t} z_{t+1},$$

for $t \geq 1, z_j \in (N \cup T)^*, 1 \leq j \leq t+1$; if $|x_{i_j}|_K = 0$ for all $1 \leq j \leq t$, then

$$y_i = z_1 x_{i_1} z_2 x_{i_2} \ldots z_t x_{i_t} z_{t+1},$$

[and $y_{i_j} = S_{i_j}, 1 \leq j \leq t$]; otherwise $y_i = x_i$. For all unspecified i we have $y_i = x_i$.

Point (i) defines a *rewriting* step (componentwise, synchronously, using one rule in all components whose current strings are not terminal); (ii) defines a *communication* step: the query symbols Q_{i_j} introduced in some x_i are replaced by the associated strings x_{i_j}, providing that these strings do not contain further query symbols. The communication has priority over rewriting (a rewriting step is allowed only when no query symbol appears in the current configuration). The work of the system is blocked when circular queries appear, as well as when no query symbol is present but point (i) is not fulfilled because a component cannot rewrite its sentential form, although it is a nonterminal string.

The relation $\Longrightarrow_r$ considered above is said to be performed in the *returning* mode: after communicating, a component resumes working from its axiom. If the brackets, [and $y_{i_j} = S_{i_j}, 1 \leq i \leq t$], are removed, then we obtain the *non-returning* mode of derivation: after communicating, a component continues the processing of the current string. We denote by $\Longrightarrow_{nr}$ the obtained relation.

The language generated by Γ is the language generated by its first component, when starting from $(S_1, \dots, S_n)$, that is

$$\begin{aligned} L_f(\Gamma) = \{w \in T^* \mid (S_1, \dots, S_n) \Longrightarrow_f^* (w, \alpha_2, \dots, \alpha_n), \\ \text{for } \alpha_i \in (N \cup T \cup K)^*, 2 \leq i \leq n\},\ f \in \{r, nr\}. \end{aligned}$$

(No attention is paid to strings in the components $2, \dots, n$ in the last configuration of a derivation; moreover, it is supposed that the work of Γ stops when a terminal string is obtained by the first component.)

A PC grammar system is said to be *centralized* if only the master component can introduce query symbols, and *non-centralized* otherwise.

PC grammar systems as above communicate *on request.* A class of parallel communicating grammar systems with communication *by command* has been considered in [2]. In such a system, each component has an associated regular language. In any moment, each component sends its current sentential form to all other components, but the transmitted string is accepted only if it is an element of the regular language associated with the receiving component. Thus, these regular languages act as filters, controlling the communication in a way similar to the control of derivations in conditional grammars.

We do not present here formally this variant of PC grammar systems and refer to [2] for details.

Example 12. Consider the PC grammar system

$$\begin{aligned} \Gamma \;=\; & (\{S_1, S_2, S_3\}, \{Q_1, Q_2, Q_3\}, \{a, b, c\}, (S_1, P_1), (S_2, P_2), (S_3, P_3)), \text{ with} \\ & P_1 = \{S_1 \to aS_1, S_1 \to aQ_2, S_2 \to bQ_3, S_3 \to c\}, \\ & P_2 = \{S_2 \to bS_2\}, \\ & P_3 = \{S_3 \to cS_3\}. \end{aligned}$$

A typical derivation in Γ proceeds as follows:

$$\begin{aligned} (S_1, S_2, S_3) & \Longrightarrow_r^* (a^n S_1, b^n S_2, c^n S_3), \text{ for some } n \geq 0, \\ & \Longrightarrow_r (a^{n+1} Q_2, b^{n+1} S_2, c^{n+1} S_3) \\ & \Longrightarrow_r (a^{n+1} b^{n+1} S_2, S_2, c^{n+1} S_2) \\ & \Longrightarrow_r (a^{n+1} b^{n+2} Q_3, bS_2, c^{n+2} S_3) \\ & \Longrightarrow_r (a^{n+1} b^{n+2} c^{n+2} S_3, bS_2, S_3) \\ & \Longrightarrow_r (a^{n+1} b^{n+2} c^{n+3}, b^2 S_2, cS_3). \end{aligned}$$

Therefore,

$$L_r(\Gamma) = \{a^n b^{n+1} c^{n+3} \mid n \geq 1\},$$

which is not a context-free language. Note that the generated language is the same in the non-returning mode (each component sends only once its string to the master) and that the system is centralized and regular.

We denote by $NCPC_nX$ the family of languages generated by PC grammar systems with at most $n \geq 1$ components, with rules of type $X \in \{REG, CF\}$, centralized (indicated by C) and non-returning (indicated by N); when non-centralized systems are used we remove the symbol C, and when returning systems are used we remove the symbol N; when no bound on the number of components is imposed we replace the subscript n by $*$.

Also in this case we only recall a few results about the generative capacity of these systems:

1. $Y_nREG - LIN \neq \emptyset, Y_nLIN - CF \neq \emptyset$, for all $n \geq 2$,
   $Y_nREG - CF \neq \emptyset$, for all $n \geq 3$, and for all $Y \in \{PC, CPC, NPC, NCPC\}$,
2. $Y_nREG - CF \neq \emptyset$, for all $n \geq 2$, and $Y \in \{NPC, NCPC\}$,
3. $LIN - (CPC_*REG \cup NCPC_*REG) \neq \emptyset$,
4. $CPC_2REG \subset CF, PC_2REG \subseteq CF$,
5. $LIN \subset PC_*REG$,
6. $CPC_*REG \subseteq MAT$,
7. $PC_*CF = NPC_*CF = RE$.

R. On the difference between context-sensitive and recursively enumerable languages. At the first sight, there is a "big difference" between the family of context-sensitive languages and that of recursively enumerable languages, between the power of grammars using erasing rules and λ-free grammars. However, if we closely examine this relationship, then we find that this difference is not so large. For instance, the following result is valid:

Theorem 9. *For every language $L \subseteq T^*, L \in RE$, there are $L' \in CS$ and $c_1, c_2 \notin T$, such that $L' \subseteq L\{c_1\}\{c_2\}^*$, and for each $w \in L$ there is $i \geq 0$ such that $wc_1c_2^i \in L'$.* (Thus, L is equal to L' modulo a tail of the form $c_1c_2^i$ for some $i \geq 0$.)

Corollary 2. (i) *Each recursively enumerable language is the projection of a context-sensitive language.*

(ii) *For each $L \in RE$ there is a language $L_1 \in CS$ and a regular language L_2 such that $L = L_1/L_2$.*

Of course, the assertions above are valid also in a "mirrored" version: with $L' \subseteq \{c_2\}^*\{c_1\}L$ in Theorem 9, and with a left quotient by a regular language in point (ii) of Corollary 2.

These results prove that the families RE and CS are "almost equal," the difference lies in a tail of arbitrary length to be added to the strings of a language; being of the form $c_1c_2^i, i \geq 1$, this tail carries no information other than its length, hence from a syntactical point of view the two languages L and L' in Theorem 9 can be considered indistinguishable.

Intuitively speaking, in order to obtain a device of the power of type-0 grammars (of Turing machines, computationally complete), it is sufficient to have (i) context-sensitivity and (ii) erasing. Of course, this is a very general formulation. For instance, we have to be careful what context-sensitivity means. As we will see when presenting the splicing operation, it is not sufficient to "sense the neighborhoud" of the site where we process a string, but to have "enough context-sensitivity to send signals at an arbitrarily long distance". In its turn, the erasing capability should be sufficient in order to be able to use an arbitrarily large workspace and to delete the useless symbols at the end of a computation.

Admitedly, these considerations are rather vague, but the reader will better understand them when looking for characterizations of recursive enumerability in any given framework (dealing with strings).

R. Universal Turing machines and type-0 grammars. A computer is a *programmable* machine, able to execute any program it receives. From a theoretical point of view, this corresponds to the notion of a *universal Turing machine*, and, in general, to the notion of a machine which is universal for a given class, in the following sense.

Consider an alphabet T and a Turing machine $M = (K, V, T, B, s_0, F, \delta)$. As we have seen above, M starts working with a string w written on its tape and reaches or not a final state (and then halts), depending on whether or not $w \in L(M)$. A Turing machine can be codified as a string of symbols over a suitable alphabet. Denote such a string by $code(M)$. Imagine a Turing machine M_u which starts working from a string which contains both $w \in T^*$ and $code(M)$ for a given Turing machine M, and stops in a final state if and only if $w \in L(M)$.

Such a machine M_u is called *universal.* It can simulate any given Turing machine, providing that a code of a particular machine is written on the tape of the universal one, together with a string to be dealt with by the particular machine.

The parallelism with a computer, as we know the computers in their general form, is clear: the code of a Turing machine is its *program*, the strings to be recognized are the input data, the universal Turing machine is the computer itself.

Let us stress here an important distinction, that between computational *completeness* and *universality*. Given a class $\mathcal{C}$ of computability models, we say that $\mathcal{C}$ is *computationally complete* if the devices in $\mathcal{C}$ can characterize the power of Turing machines (or of any other type of equivalent devices). This means that given a Turing machine M we can find an element C in $\mathcal{C}$ such that C is equivalent with M. Thus, completeness refers to the capacity of covering the level of computability (in grammatical terms, this means to generate all recursively enumerable languages). Universality is an internal property of $\mathcal{C}$ and it means the existence of a fixed element of $\mathcal{C}$ which is able to simulate any given element of $\mathcal{C}$, in the way described above for Turing machines.

The idea of a universal Turing machine was introduced by Turing himself, who has also produced such a machine. Many universal Turing machines are now available in the literature.

Given a Turing machine M we can effectively construct a type-0 grammar G such that $L(G) = L(M)$ as follows. Take a Turing machine $M = (K, V, T, B, s_0, F, \delta)$ and construct a non-restricted Chomsky grammar G working as follows: starting from its axiom, G nondeterministically generates a string w over V, then it makes a copy of w (of course, the two copies of w are separated by a suitable marker; further markers, scanners and other auxiliary symbols are allowed, because they can be erased when they are no longer necessary). On one of the copies of w, G can simulate the work of M, choosing nondeterministically a computation as defined by δ; if a final state is reached, then the witness copy of w is preserved, and everything else is erased.

Details of such a construction can be found in [9], as well as in the prerequisites chapter of [6].

Applying this construction to a universal Turing machine M_u, we obtain a *universal type-0 Chomsky grammar* G_u, a grammar which is universal in the following sense: the language generated by G_u consists of strings of the form, say, $w\#code(M)$, such that $w \in L(M)$. (We can call the language $\{w\#code(M) \mid w \in L(M)\}$ itself universal, and thus any grammar generating this language is universal.) However, we are interested in a "more grammatical" notion of universality, and this leads to the following definition.

A triple $G = (N, T, P)$, where the components N, T, P are as in a usual Chomsky grammar, is called a *grammar scheme*. For a string $w \in (N \cup T)^*$ we define the language $L(G, w) = \{x \in T^* \mid w \Longrightarrow^* x\}$, the derivation being performed according to the productions in P.

A *universal type-0 grammar* is a grammar scheme $G_u = (N_u, T_u, P_u)$, where N_u, T_u are disjoint alphabets, and P_u is a finite set of rewriting rules over $N_u \cup T_u$, with the property that for any type-0 grammar $G = (N, T_u, S, P)$ there is a string $w(G)$ such that $L(G_u, w(G)) = L(G)$.

Therefore, the universal grammar simulates any given grammar, providing that a code $w(G)$ of the given grammar is taken as a starting string of the universal one.

There are universal type-0 grammars in the sense specified above. A concrete construction can be found in Chapter 3 of [6].

A universal grammar G_u can be constructed by encoding in the "type-0 grammars programming language" the very way of using a grammar in a derivation process: choose a rule, remove an occurrence of its left hand member, introduce instead an occurrence of its right hand member, check whether or not a terminal string is obtained.

For the membrane computing area it is important to note that when a characterization of recursively enumerable languages/numbers is obtained in a constructive manner (with the construction started from Turing machines or from Chomsky type-0 grammars), the universality is transferred "for free" from Turing machines/type-0 grammars to membrane systems (the same assertion is true for any type of computing devices). Thus, computational completeness directly means universality, hence programability. (There is a question here, about what means the code of a membrane system and how it is introduced in an universal system; the answer is always provided by the construction from the proof of the completeness result.)

Another important observation is that there is no natural definition of universality, hence no universality result, for levels of the Chomsky hierarchy other than the type-0 grammars (Turing machines). This implies the fact that when having a new computing machinery, in order to have a programmable device, we need a machinery which is computationally complete (hence containing "enough" context-sensitivity and "sufficient" erasing possibilities).

A. Splicing, insertion-deletion, context adjoining. Rewriting is only one (fundamental and well-known) operation by which we can process strings (in particular, in the membrane computing area). In formal language theory there are many other string operations, some of them inspired from the very area where the membrane computing has its origins, the biology. This is the case with the splicing operation, considered in 1987 by T. Head, as a formal counterpart of the recombination of DNA molecules under the influence of restriction enzymes. Comprehensive details can be found in [6], hence we recall here only a few ideas.

Consider an alphabet V and two symbols $\#, \$$ not in V. A *splicing rule* over V is a string $r = u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in V^*$. For such a rule r and for $x, y, w, z \in V^*$ we define

$$\begin{aligned}(x, y) \vdash_r (w, z) \quad \text{iff} \quad & x = x_1u_1u_2x_2, \;\; y = y_1u_3u_4y_2, \\ & w = x_1u_1u_4y_2, \;\; z = y_1u_3u_2x_2, \\ & \text{for some } x_1, x_2, y_1, y_2 \in V^*.\end{aligned}$$

(One cuts the strings x, y in between u_1, u_2 and u_3, u_4, respectively, and one recombines the fragments obtained in this way.)

Based on this operation, one can define a language generating device as follows. First, the operation is extended to languages. An *H scheme* is a pair $\sigma = (V, R)$, where V is an aphabet and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules. For an H scheme σ and a language $L \subseteq V^*$ we define

$$\begin{aligned}
\sigma(L) &= \{z \in V^* \mid (x, y) \vdash_r (w, z) \text{ or } (x, y) \vdash_r (z, w), \\
&\quad \text{for some } x, y \in L, \text{ and } r \in R\}, \\
\sigma^0(L) &= L, \\
\sigma^{i+1}(L) &= \sigma^i(L) \cup \sigma(\sigma^i(L)), \text{ for } i \geq 0, \\
\sigma^*(L) &= \bigcup_{i \geq 0} \sigma^i(L).
\end{aligned}$$

(Thus, $\sigma^*(L)$ is the closure of L under the splicing with respect to the rules from R.)

An *extended H system* is a construct $\gamma = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$ (terminal alphabet), $A \subseteq V^*$ (axioms), and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules over V. The language generated by γ is defined by

$$L(\gamma) = \sigma^*(A) \cap T^*,$$

where $\sigma = (V, R)$ is the underlying H scheme associated with γ.

For two families FL_1, FL_2 of languages, we denote by $EH(FL_1, FL_2)$ the family of languages $L(\gamma)$ generated by H systems $\gamma = (V, T, A, R)$ with $A \in FL_2$ and $R \in FL_2$ (note that the splicing rules are written as strings, hence it makes sense to speak about the type of the "language" L in a given hierarchy of languages).

The following results are basic in the splicing area:

1. $EH(REG, FIN) = REG$, $EH(CF, FIN) = CF$.
2. $EH(FIN, REG) = RE$.

Therefore, using a finite set of splicing rules, when starting from a regular language we get only regular languages. This is a good illustration of the discussion about the context-sensitivity and the erasing needed for obtaining the power of Turing machines: splicing is context-sensitive, because we cut the strings in a given context, it also has erasing, but they are not sufficient. For instance, it is clear that by cutting and recombining fragments of strings we loose information about the fragments, we cannot send signals at long distance.

However, there are more than one dozen of controlled H systems, with rather weak regulations about the use of the splicing rules, which can characterize *RE*; the feeling is that one needs rather weak additional context-sensitivity features in order to lead H systems to computational universality (and this is the case also when using the splicing in the P systems area).

Other operations with strings investigated in formal language theory mainly with a motivation from linguistics are those dealing with inserting or adjoining strings or pairs of strings to current strings; dually, single strings or couples of strings can be deleted from current strings.

For instance, an insertion rule is of the form (u, x, v), with the meaning that a string $w = w_1uvw_2$ is transformed by such a rule to $w' = w_1uxvw_2$ (the string x was inserted in the context (u, v)). A rule of the same form can be used in the deletion manner: from $z = z_1uxvz_2$ we pass to $z' = z_1uvz_2$ (the substring x was deleted from the context (u, v)). Now, an *insertion-deletion system* can be defined as a construct $\gamma = (V, T, A, I, D)$, where V is an alphabet, $T \subseteq V$ (terminal alphabet), $A \subseteq V^*$ (a finite set of axioms), and I, D are finite sets of insertion and deletion rules, respectively. The generated language consists of all strings over T^* which can be obtained by starting from the strings from A and using iteratively rules from I and from D.

A variant of interest is that of *Marcus contextual grammars*, where one deals with pairs of strings, inserted or deleted from current strings. We only present here the so-called *internal contextual grammars with selection*, which are constructs of the form $G = (V, A, P)$, where V is an alphabet, A is a finite set of strings over V (axioms), and P is a finite set of pairs of the form $(C, (u, v))$, with $C \subseteq V^*$ and $(u, v) \subseteq V^* \times V^*$; C is called the *selector* and (u, v) is called the *context* of the *rule* $(C, (u, v))$.

For two strings $w, z \in V^*$ we write $w \Longrightarrow z$ if $w = w_1xw_2, z = w_1uxvw_2$, for some $(C, (u, v) \in P$, with $x \in C$, and $w_1, w_2 \in V^*$ (the context (u, v) is adjoined to the string x from the selector C associated with the context). The language generated by G is defined by

$$L(G) = \{z \in V^* \mid w \Longrightarrow^* z \text{ for some } w \in A\}.$$

Details about insertion-deletion systems and about Marcus contextual grammars (of various forms: for instance, in the *external* variant, the contexts are always adjoined at the ends of the processed string) can be found in many places, in particular, in [5] and [8].

References

[1] E. Csuhaj-Varju, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.

[2] E. Csuhaj-Varju, J. Kelemen, Gh. Păun: Grammar systems with WAVE-like communication, *Computers and AI*, 15, 5 (1996), 419–436.

[3] J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.

[4] R. Freund, Gh. Păun, On the number of non-terminals in graph-controlled, programmed, and matrix grammars, *Proc. MCU Conf.*, Chişinău, 2001, *Lecture Notes in Computer Science*, 2055, Springer-Verlag, 2001, 214–225.

[5] Gh. Păun, *Marcus Contextual Grammars*, Kluwer, Dordrecht, Boston, 1997.

[6] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Heidelberg, 1998.

[7] G. Rozenberg, A. Salomaa, *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.

[8] G. Rozenberg, A. Salomaa, Eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.

[9] A. Salomaa, *Formal Languages*, Academic Press, New York, 1973.

Index of Notions