

Comparison between Free and Regular Languages

Translated and adapted by L. Breveglieri

GRAMMARS FOR REGULAR LANGUAGES

Regular languages are a special case of the free ones

Regular languages can be generated by grammars that exhibit strongly restricted rules

The phrases of a regular language necessarily contain repeated factors as the length of the string grows

It is possible to prove in a rigorous way that some (free) languages cannot be generated by any regular expression

REGULAR LANGUAGES

Recognizing phrases requires only a finite amount of memory

FREE LANGUAGES

Recognizing phrases requires an unbounded amount of memory

FROM THE REGULAR EXPRESSION TO THE FREE GRAMMAR

The iterative regular operators (star and cross) must be replaced with recursive rules

Decompose the regexp into subexpressions and denumerate them progressively. After the very definition of regexp, the following cases are obtained, which allow Us to design the corresponding rules shown aside

1. $r = r_1.r_2....r_k$
2. $r = r_1 \cup r_2 \cup \cup r_k$
3. $r = (r_1)^*$
4. $r = (r_1)^+$
5. $r = b \in \Sigma$
6. $r = \varepsilon$

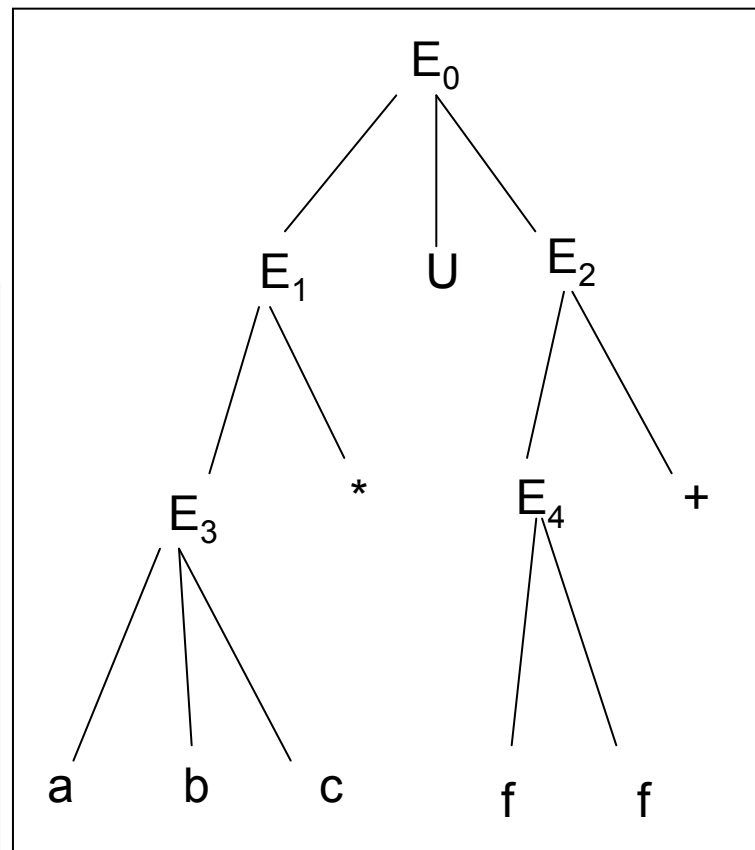
1. $E = E_1E_2...E_k$
2. $E = E_1 \cup E_2 \cup ... \cup E_k$
3. $E = EE_1 \mid \varepsilon$ or $E = E_1E \mid \varepsilon$
4. $E = EE_1 \mid E_1$ or $E = E_1E \mid E_1$
5. $E = b$
6. $E = \varepsilon$

Uppercase letters indicate the non-terminal symbols. If E_i is a terminal symbol $a_i \in \Sigma$, the symbol a_i is written instead of E_i

EXAMPLE

$$E = (abc)^* \cup (ff)^+$$

- | | | |
|----|----------------|--|
| 2. | $E_1 \cup E_2$ | $E_0 \rightarrow E_1 \mid E_2$ |
| 3. | E_3^* | $E_1 \rightarrow E_1 E_3 \mid \varepsilon$ |
| 4. | E_4^+ | $E_2 \rightarrow E_2 E_4 \mid E_4$ |
| 1. | abc | $E_3 \rightarrow abc$ |
| 1. | ff | $E_4 \rightarrow ff$ |



If the regexp to start from is ambiguous, the derived equivalent free grammar is ambiguous as well

Every regular language is free, but there are many free languages that are not regular, like for instance the palindromes and the arithmetic expressions with parentheses

$$REG \subset LIB$$

LINEAR GRAMMAR

A grammar is said to be LINEAR if every rule has at most one non-terminal symbol in the right member of the rule

$$A \rightarrow uBv \text{ where } u, v \in \Sigma^*, B \in (V \cup \varepsilon)$$

The family of the linear free languages is still by far larger than that of the regular languages

EXAMPLE: here follows a linear free language that is not regular

$$L_1 = \{a^n b^n \mid n \geq 1\} = \{ac, aacc, \dots\}$$
$$S \rightarrow aSc \mid ac$$

UNI-LINEAR GRAMMAR (or GRAMMAR of TYPE 3)

RIGHT UNI-LINEAR RULE (or simply RIGHT LINEAR)

$$A \rightarrow uB \quad \text{where} \quad u \in \Sigma^*, B \in (V \cup \varepsilon)$$

LEFT UNI-LINEAR RULE (or simply LEFT LINEAR)

$$A \rightarrow Bv \quad \text{where} \quad v \in \Sigma^*, B \in (V \cup \varepsilon)$$

The corresponding syntax trees grow in a heavily unbalanced way

If a left or right uni-linear grammar contains a recursive derivation, then such a derivation is recursive on the left or right, respectively

EXAMPLE: the phrases that contain the substring *aa* and terminate with *b* are generated by the following regexp (ambiguous)

$$(a \mid b)^* aa(a \mid b)^* b$$

1. right linear grammar G_d

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

2. left linear grammar G_s

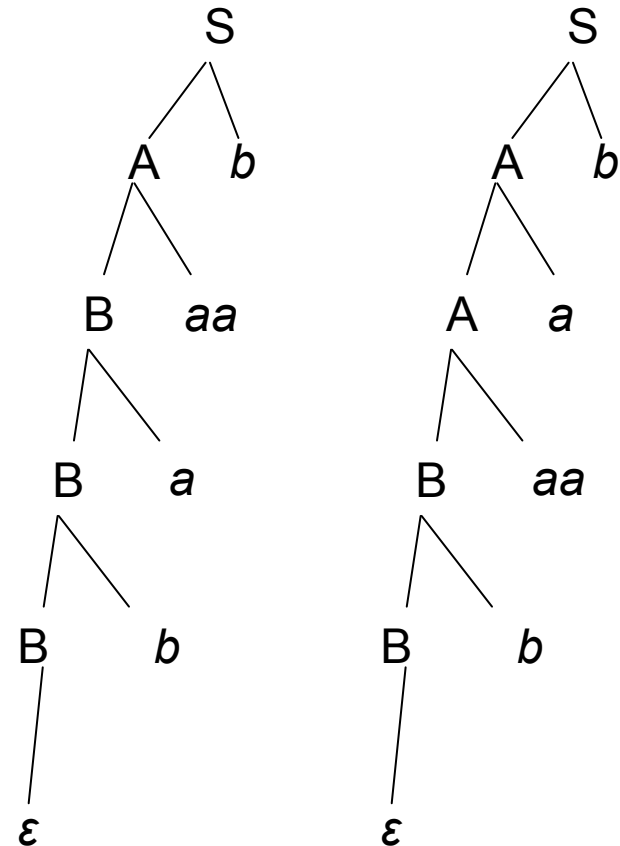
$$S \rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa$$

$$B \rightarrow Ba \mid Bb \mid \varepsilon$$

3. equivalent non-unilinear grammar

$$E_1 \rightarrow E_2 aa E_2 b \quad E_2 \rightarrow E_2 a \mid E_2 b \mid \varepsilon$$

G_s – syntax trees of the ambiguous phrase *baaab*



EXAMPLE : arithmetic expressions without parentheses

$$L = \{a, a + a, a * a, a + a * a, \dots\}$$

$$G_d : S \rightarrow a \mid a + S \mid a * S$$

$$G_s : S \rightarrow a \mid S + a \mid S * a$$

G_s and G_d are not adequate to compilation, as their syntax structure does not reflect the usual precedence rules between arithmetic operators. It is convenient to transform a unilinear grammar into a strictly unilinear grammar, where every rule contains at most a single terminal character

$$A \rightarrow aB \quad \text{or} \quad A \rightarrow Ba \quad \text{where} \quad a \in (\Sigma \cup \varepsilon), B \in (V \cup \varepsilon)$$

Without any loss of generality, one can impose to have only null terminal rules

$$A \rightarrow aB \mid \varepsilon \quad \text{where} \quad a \in \Sigma$$

FROM UNILINEAR GRAMMAR TO REGULAR EXPRESSION: LINEAR EQUATIONS

UNILINEAR GRAMMAR / REGULAR LANGUAGE

First it is proved that the family of the languages generated by unilinear grammars strictly contains that of regular languages. Then it is proved that these two families actually coincide

One can think of the rules of a right (uni)linear grammar as of equations, the unknown terms of which are the languages generated by each non-terminal symbol.

Suppose that G is a strictly right linear grammar and that G contains only null terminal rules $A \rightarrow \varepsilon$

$$G = (V, \Sigma, P, S)$$
$$L_A = \left\{ x \in \Sigma^* \mid A \xRightarrow{+} x \right\} \quad L(G) \equiv L_S$$

replace the (recursive) rule $A \rightarrow a A$ with the equation $L_A = L_a L_A$

replace the rule $A \rightarrow b B$ with the equation $L_A = L_b L_B$

replace the rule $A \rightarrow a$ with the equation $L_A = L_a$

unite terminal rules to non-terminal rules:

$A \rightarrow b B$ and $A \rightarrow c$ so that $L_A = L_c \mid L_b L_B$

and so on when there are more terms

then solve the equations by substitution

in the case of a recursive equation $L_A = L_x \mid L_y L_A$,
solve it by means of the Arden identity: $L_A = L_y^* L_x$

EXAMPLE: set-theoretic equations

Here follows a grammar that generates a list of elements e (possibly empty) separated by the character s

$$S \rightarrow sS \mid eA \quad A \rightarrow sS \mid \varepsilon$$

The grammar can be transformed into a set of equations, as aside

EVERY UNILINEAR LANGUAGE
IS REGULAR

$$\begin{cases} L_s = sL_s \cup eL_A \\ L_A = sL_s \cup \varepsilon \end{cases}$$

$$\begin{cases} L_s = sL_s \cup e(sL_s \cup \varepsilon) \\ L_A = sL_s \cup \varepsilon \end{cases}$$

$$\begin{cases} L_s = (s \cup es)L_s \cup e \\ L_A = sL_s \cup \varepsilon \end{cases}$$

and by using Arden identity one obtains

$$\begin{cases} L_s = (s \cup es)^* e \\ L_A = sL_s \cup \varepsilon \quad L_A = s(s \cup es)^* e \cup \varepsilon \end{cases}$$

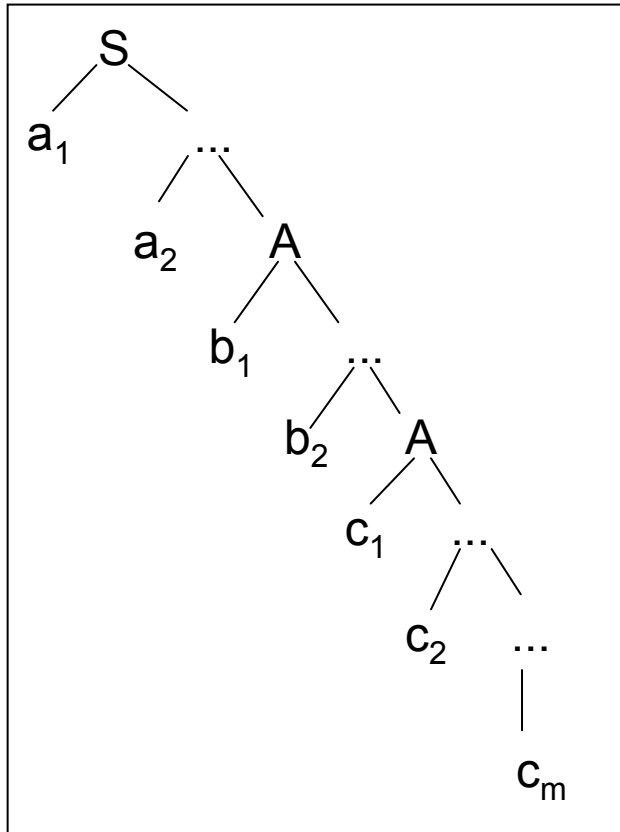
COMPARISON BETWEEN REGULAR AND FREE LANGUAGES: UNAVOIDABLE REPETITIONS

THE LIMITATIONS OF THE FAMILY OF REGULAR LANGUAGES:
which kinds of linguistic structures can be modeled with regular expressions,
and which ones instead need the larger generating power of free grammars ?

Only thanks to recursion, a grammar will generate an infinite language.
Here follows a property useful to prove that certain free languages
are not regular

PROPERTY: Let G be a unilinear grammar. Every sufficiently long phrase x (that is, $|x| > k$ for a constant k that depends only on G), generated by G , can be factored as $x = t u v$, where u is not empty, in such a way that for every integer $n \geq 1$, the string $t u^n v$ is generated by G as well (one says that the phrase x can be pumped by inserting in the middle new occurrences of the factor u , as many times as one wants)

PROOF: Consider a strictly right linear grammar, that has exactly k non-terminal symbols. In the derivation of a long phrase x , of k or more characters, there necessarily is a non-terminal symbol A that will occur twice or more times



$$t = a_1 a_2 \dots, \quad u = b_1 b_2 \dots \quad v = c_1 c_2 \dots c_m$$

$$S \xRightarrow{+} tA \xRightarrow{+} tuA \xRightarrow{+} tuv$$

allows to derive also $tuuv \quad tv$

TRY TO USE THE ABOVE PROPERTY
TO PROVE THAT L_1 IS NOT REGULAR

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

EXAMPLE: language with two identical exponents

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

Suppose by refutation that the language is regular

$$x = a^k c^k = tuv \quad \text{where } u \text{ is not empty}$$

t	u	v	iterated string
1. $t = a^h$	$u = a^i$	$v = a^j c^k$	$a^h a^i a^i a^j c^k$
	$1 \leq i \leq k$	$h + i + j = k$	$h + 2i + j > k$
2. $t = a^h$	$u = a^i c^j$	$v = c^m$	$a^h a^i c^j a^i c^j c^m \notin a^+ c^+$
	$1 \leq i, j \leq k$	$h + i = j + m = k$	
3. $t = a^k c^h$	$u = c^i$	$v = c^j$	$a^k c^h c^i c^i c^j$
	$1 \leq i \leq k$	$h + i + j = k$	$h + 2i + j > k$

In all the possible cases, the iterated string (that should belong to the language if this were regular), does not exhibit the required form $a^n c^n$

THE ROLE OF SELF-INCLUSIVE DERIVATIONS

$$A \overset{+}{\Rightarrow} uAv \quad u \neq \varepsilon \wedge v \neq \varepsilon$$

A self-inclusive derivation may not occur in the unilinear grammar of regular languages. This does not happen in the free grammars that generate free non-regular languages (palindromes, Dyck, etc)

The absence of self-inclusive derivations allows to solve the set-theoretic equations associated with unilinear grammars, and gives rise to the simple structure of the languages that such grammars generate

A GRAMMR THAT DOES NOT ADMIT ANY SELF-INCLUSIVE DERIVATION ALWAYS GENERATES A REGULAR LANGUAGE

EXAMPLE: a non-self-inclusive grammar

ALTHOUGH THE GRAMMAR CONTAINS
SELF-INCLUSIVE DERIVATIONS, THE
GENERATED LANGUAGE MAY BE
REGULAR, IN SOME SPECIAL CASES



ONE-LETTER LANGUAGES
ENJOY A CURIOUS PROPERTY



THE LANGUAGE DEFINED BY A FREE
GRAMMAR OVER A ONE-LETTER
TERMINAL ALPHABET ($|\Sigma| = 1$) IS
REGULAR

$$G: \quad S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \varepsilon$$

$$\begin{cases} L_s = L_A L_s \cup bL_A \\ L_A = aL_A \cup \varepsilon \end{cases}$$

$$\begin{cases} L_s = L_A L_s \cup bL_A \\ L_A = a^* \end{cases}$$

$$L_s = a^* L_s \cup ba^*$$

$$L_s = (a^*)^* ba^* \quad L(G) = a^* ba^*$$

$$G = \{S \rightarrow aSa \mid \varepsilon\}$$

$$S \Rightarrow aSa \quad L(G) = (aa)^*$$

can be defined as

$$G_d = \{S \rightarrow aaS \mid \varepsilon\}$$

THE LIMITATIONS OF FREE LANGUAGES

In the free languages, the sufficiently long phrases necessarily contain some substrings that can be repeated as many times as one wants, and hence that give rise to nested structures. Such a constraint prevents free grammars of generating phrase structures where three or more parts are repeated for the same number of times

PROPERTY: Let G be a free grammar. Every sufficiently long phrase x (that is, $|x| > k$ for a constant k that depends only on G), generated by G , can be factored as $x = y u w v z$, where u, v are not empty, in such a way that for every integer $n \geq 1$, the string $y u^n w v^n z$ is generated by G as well (one says that the phrase x can be pumped by inserting in the middle equal numbers of new occurrences of the factors u, v , as many times as one wants)

THE LIMITATIONS OF FREE LANGUAGES

EXAMPLE: THE LANGUAGE WITH THREE EXPONENTS IS NOT FREE
(the proof exploits the possibility of pumping a long phrase by repeating two independent factors, which is incompatible with the definition of such a language)

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

EXAMPLE: THE COPY LANGUAGE

This sub-language is frequent in the programming languages, whenever two lists need to have identical matching elements, like for instance in the lists of the formal and actual parameters of a procedure or function declaration and call

$$L_{replic} = \{uu \mid u \in \Sigma^+\}$$
$$\Sigma = \{a, b\} \quad x = abbbabbb$$

THE CLOSURE PROPERTIES OF THE REG AND LIB FAMILIES

Here the closure properties of the REG and LIB families of languages are examined, with respect to the most common language operators

Let L and R be two languages belonging to LIB and REG, respectively

$R^R \in REG$	$L^R \in LIB$	denotes both union and concatenation
$R^* \in REG$	$L^* \in LIB$	
$\neg R \in REG$	$\neg L \notin LIB$	denotes both union and concatenation
$R_1 \oplus R_2 \in REG$	$L_1 \oplus L_2 \in LIB$	
$R_1 \cap R_2 \in REG$	$L_1 \cap L_2 \notin LIB$	
	$L \cap R \in LIB$	

COMMENTS AND EXAMPLES

- 1) The mirror image of $L(G)$ is generated by the mirror grammar, obtained from G by mirroring the right sides of the rules (a right linear G becomes left linear)
- 2) The star, union and concatenation of free languages are themselves free

G_1	$G_2, L_1 \ L_2, \text{ axioms } S_1 \ S_2, V_1 \cap V_2 = \emptyset$
star	$S \rightarrow SS_1 \mid \varepsilon$
union	$S \rightarrow S_1 \mid S_2$
concatenation	$S \rightarrow S_1 S_2$

- 3) If the two grammars to be united share some non-terminal names, their union will generate a superset language of the true union of the two languages

$$\begin{aligned} G_1 &= \{S_1 \rightarrow aA, A \rightarrow bA \mid b\}, \quad G_2 = \{S_2 \rightarrow aA, A \rightarrow cA \mid c\} \\ G_{1 \cup 2} &= \{S \rightarrow S_1 \mid S_2, S_1 \rightarrow aA, A \rightarrow bA \mid b, S_2 \rightarrow aA, A \rightarrow cA \mid c\} \\ a(b \mid c)^+ &\supset L(G_1) \cup L(G_2) = ab^+ \cup ac^+ \end{aligned}$$

4) The intersection of two free languages, in general, is not a free language

$$\left\{a^n b^n c^n \mid n \geq 1\right\} = \left\{a^n b^n c^+ \mid n \geq 1\right\} \cap \left\{a^+ b^n c^n \mid n \geq 1\right\}$$

5) There exist free languages the complements of which (w.r.t. the universal language over the same alphabet) are not free languages

CAUTION: of course, there exist also free languages the complements of which are still free languages (and similarly for intersection)

INTERSECTION OF FREE LANGUAGES WITH REGULAR LANGUAGES

One way to make a free grammar more selective, is to restrict the generated free language through a regular filter

EXAMPLE: regular filter over the Dyck language - $\Sigma = \{ a, c \}$ (a open c close)

$$\begin{aligned} L_1 &= L_D \cap \neg(\Sigma^* cc \Sigma^*) = (ac)^* \\ L_2 &= L_D \cap \neg(\Sigma^* ca \Sigma^*) = \{ a^n c^n \mid n \geq 0 \} \end{aligned}$$

The former intersection filters the Dyck language by removing the phrases that contain the factor cc , that is the phrases containing nested structures

The latter intersection filters the Dyck language by removing the phrases that contain two or more concatenated nested structures, leaving only the phrases with one nested structure

Both filtered languages are free, and the former one is regular as well

ALPHABETIC TRANSFORMATIONS

Very similar languages may differ only as for a few terminal symbols
TRANSLITTERATION (or ALPHEBETIC HOMOMORPHISM) is a linguistic operation that replaces some terminal characters by some other terminal characters, over the same or a different alphabet. Some characters can also be canceled (that is, replaced by the empty string ε)

TRANSLITTERATION (ALPHABETIC HOMOMORPHISM) is a function

$$h : \Sigma \rightarrow \Delta \cup \{\varepsilon\}$$

ε -free translitteration (or alpha. hom.): for every $c \in \Sigma$

$$h(c) \neq \varepsilon$$

Translitteration of a source string

$$\begin{aligned} a_1 a_2 \dots a_n &\in \Sigma^* \\ h(a_1) h(a_2) \dots h(a_n) \\ h(v.w) &= h(v).h(w) \end{aligned}$$

EXAMPLE: printer

$$h(c) = c \quad \text{if } c \in \{a, b, \dots, z, 0, 1, \dots, 9\};$$

$$h(c) = c \quad \text{if } c \text{ is a punctuation character}$$

$$h(c) = \pi \quad \text{if } c \in \{\alpha, \beta, \dots, \omega\};$$

$$h(\text{start-text}) = h(\text{end-text}) = \varepsilon$$

$$h(\text{start-text} \quad \textit{cost } \pi \textit{ is } 3.14 \quad \text{end-text}) = \textit{cost } \pi \textit{ is } 3.14$$

WORD TRANSLITTERATION (of ARBITRARY HOMOMORPHISM)

the transformed image of a character may be a string longer than one

$$h(\leftarrow) ::= h(c) = c \quad \text{for every other } c \in \Sigma$$

SUBSTITUTION: each character of the source string can be replaced by an entire string, out of a given language; the language to take the string from depends on the character to be replaced

For instance, given the following source alphabet: $\Sigma = a, b, \dots$

a substitution h associates with each letter a a specific replacement language over the destination alphabet Δ

$$h(a) = L_a, \quad h(b) = L_b$$

Applying the substitution h to a source string x

$$x = a_1 a_2 \dots a_n \in \Sigma^*$$

yields the following set of strings $h(x) = \{ y_1 y_2 \dots y_n \mid y_i \in L_{a_i} \}$

A word translitteration is a special case of substitution: each replacement language contains only one string. A translitteration is a special case of substitution: each replacement language reduces to one character (or the empty string)

CLOSURE WITH RESPECT TO ALPHABETIC TRANSFORMATIONS

PROPERTY: Both families REG and LIB (of regular and free languages, resp.) are closed with respect to word translitteration

PROOF: Let G be a grammar that generates L , and h a word translitteration from the source alphabet Σ to the destination alphabet Δ . Apply the transformation h to every rule $A \rightarrow \alpha$ of G , and leave unchanged the arrow metasymbol \rightarrow and every non-terminal symbol

$$h(A \rightarrow \alpha) \equiv A \rightarrow h(\alpha)$$

The rules that are obtained in this way, are still of the free type and closely resemble those they derive from (only the terminal symbols are changed)

The obtained grammar generates the transformed language $h(L(G))$

CLOSURE WITH RESPECT TO ALPHABETIC TRANSFORMATIONS

PROPERTY: Both families REG and LIB (of regular and free languages, resp.) are closed with respect to the substitution with languages of the same family

PROOF: Let G be a grammar that generates the language L , and let h be a substitution such that, for every character $c \in \Sigma$, the replacement language L_c is free and is generated by a grammar G_c with axiom S_c ; moreover, assume that the non-terminal alphabets of the grammars G, G_a, G_b, \dots are disjoint

In order to obtain a grammar G' that generates the transformed language $h(L)$, apply the alphabetic transliteration f to the rules of G , where f is defined as $f(c) = S_c$ for every terminal character $c \in \Sigma$; and leave $f(A) = A$ for any other symbol of G

To obtain the grammar G'

- to every rule $A \rightarrow \alpha$ of G apply the transliteration f that replaces each terminal character c of G by means of the axiom S_c of the corresponding grammar
- to the obtained rules, add those of the grammars G_a, G_b, \dots
the obtained final grammar generates the transformed language $h(L(G))$