RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

L. BREVEGLIERI

Fair expressions and regular languages over lists

RAIRO. Informatique théorique et applications, tome 31, n° 1 (1997), p. 15-66

<http://www.numdam.org/item?id=ITA_1997__31_1_15_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 31, n° 1, 1997, pp. 15-66)

FAIR EXPRESSIONS AND REGULAR LANGUAGES OVER LISTS (*) (**)

by L. BREVEGLIERI

Abstract. – *In this paper a formalism is proposed, named fair expressions, partly introduced in* [Bre94], *that extends regular expressions to lists, having strings as components. This formalism uses classical regular operators,* i.e. *catenation and its closure, and novel ones, namely the operator of merge and its closure, which are natural for lists. Fair expressions allow to define languages of lists, named fair languages, which can be compared to word languages by flattening the lists into strings. In this paper the basic properties of fair languages are briefly summarized and also extended with respect to previous works* [Bre94]*: hierarchy, semilinearity, closure, decidability and comparison with the Chomsky hierarchy are dealt with. The family of fair languages is however far larger than the regular one; as a novel contribution this paper investigates its subfamilies that are comparable with regular languages. The main result is that the regular subfamilies of fair languages constitute a proper hierarchy. These subfamilies are then characterized and their properties are explored, showing that they are, in general, more mathematically tractable than fair languages. The conclusion lists comparisons with related works, open problems and research directions.*

Résumé. – *Cet article propose un formalisme, appelé expressions équitables, qui a été partiellement introduit en* [Bre94], *et qui étend les expressions régulières aux listes dont les composantes sont des mots. Ce formalisme utilise les opérateurs rationnels classiques, c'est-à-dire la concaténation et l'étoile, ainsi que de nouveaux opérateurs tels que celui de fusion et sa fermeture, qui sont des opérateurs naturels sur les listes. Les expressions équitables permettent de définir des languages de listes appelés languages équitables qui peuvent être comparés aux langages de mots, les listes une fois converties en mots par concaténation des composantes. Dans cet article les propriétés fondamentales des languages équitables sont brièvement rappelées et généralisées par rapport aux travaux antérieurs* [Bre94]*: hiérarchie, semilinéarité, clôture, décidabilité et comparaison avec la hiérarchie de Chomsky sont traités. Cette famille contient strictement celle des langages réguliers; l'étude des sous-familles comparables à celle des langages réguliers est une contribution originale de ce travail. Le résultat principal est que ces sous-familles régulières de langages équitables forme une hiérarchie propre. Ces sous-familles ensuite caractérisées et leurs propriétés sont explorées, et nous montrons ainsi qu'elles sont mathématiquement plus manipulables que celle des langages équitables. En conclusion nous mentionnons des travaux qui sont reliés à notre recherche, nous posons des problèmes ouverts et indiquons des directions de recherche.*

(*) Received May 1996, accepted February 1997.

(**) Work partially supported by a grant of the "Ministero dell'Università e della Ricerca Scientifica e Tecnologica" (MURST 60%) (Ministery of the University and of the Scientific and Technological Research), Italy, and by a grant of ESPRIT-BRA "Algebraic and Syntactic Methods In Computer Science II", (ASMICS II), Special Contract n° 6317, European Union.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/97/01/$ 7.00

1. INTRODUCTION

In order to extend the modeling capacity of formal language theory to some complex phenomena, we consider in this work *lists* of strings and sets of lists or *list languages*. A list is an ordered sequence of strings, separated by delimiters. We propose a new approach for defining fair expressions and list languages, similar to the use of regular expressions for defining regular languages. To this purpose the classical operators of string catenation and its closure (the Kleene star) are extended to lists and two new natural operators over lists are introduced: the *merge* operator and its closure. Catenation builds a new list by appending two lists, as usuals, while merge builds a new list by ordely interleaving the components of two lists.

A list of strings can be interpreted as the trace of a concurrent system: the letters represent atomic actions, each component (a string) represents a sequence of actions executed in a single time slot and the whole list (a sequence of strings) represents a sequence of time slots. In [Bre93, Bre94, Fri94] it is proved that the list languages generated by fair expressions coincide with the sets of the traces representing parallel programme schemes. Fair expressions were introduced in [Bre94] (a complete treatment can be found in [Bre93, Fri94]), but a summary of their definition and properties, with some extensions, is included in this paper for completeness. The paper is focused instead on the comparison between fair languages and regular (or rational) ones.

Since a list of strings separated by a delimiter can be viewed itself as a single string defined over an alphabet extended with the delimiter (the "word image" of the list), a natural definition of regularity can be given for lists: a list language is regular if and only if its word image is regular. A second, but weaker, definition of regularity for lists is based on the notion of free segmentation of the strings belonging to a regular language. This smaller family of regular list languages is named simple regular list languages. The central result is that the three families of simple regular, regular and fair list languages are strictly included, thus forming a proper hierarchy.

Several closure properties of fair languages and (simple) regular list languages are then investigated. It is shown that by combining (simple) regular list languages via various operations (homomorphism, replication, catenation, star, etc.) one obtains intermediate families of list languages. Standard language decision problems (membership, emptiness, equivalence, containment, etc.) are investigated for fair languages and (simple) regular list languages. This completes the picture of the properties of fair languages

and their regular subfamilies. Finally, comparisons with existing language models, e.g. the Chomsky hierarchy, and alternative formalizations help in understanding the generative power of lists.

The paper is organized as follows. In section 2 the definitions and the operations on lists are summarized. In section 3 fair expressions, and fair and regular list languages are defined. In section 4 simple examples of list languages are presented, and in section 5 a more complex example is discussed. In section 6 a hierarchy property is proved. In section 7 the general properties of the introduced families of list languages are proved: semilinearity, closure and decidability. In section 8 a comparison with the Chomsky hierarchy and AFL is carried out. In section 9 fair expressions are partially formalized in an algebraic way. Finally, in section 10 open problems and research directions are discussed.

2. LISTS OF STRINGS

Lists, having strings as their elements, are objects similar, under many respects, to strings. Lists are sequences of strings, separated by delimiters, whereas strings are sequences of characters. We extend the string operations, e.g. catenation and Kleene star, to the lists, also introducing a new operator and its closure; this operator is named merge and is natural for lists.

2.1. Definition of list

A *list* $\underline{x}$ over some non-void finite alphabet Σ is a linearly ordered set of strings (also including the empty string ε), and is denoted as:

$$\underline{x} = [x_1;\ x_2;\ \ldots;\ x_n] \qquad x_i \in \Sigma^* \quad \text{for } 1 \leq i \leq n$$

Note that we always assume that $(;) \notin \Sigma$; the special character $(;)$ is a separator, not a letter of the alphabet Σ. The string $x_i \in \Sigma^*$, with $1 \leq i \leq n$, is said to be the i^{th} *component* of the list $\underline{x}$. List names are usually underlined, to distinguish them from strings; the same notation is frequently used to distinguish list sets from word sets.

The *null* list, containing no components, is denoted [ ]; it must not be confused with the list $[\varepsilon]$, where ε is the empty string, because the list $[\varepsilon]$ contains one component. The sets of all the lists are denoted as follows, similarly to strings. Given an alphabet Σ, the two sets of lists:

$$\underline{\Sigma}^{[*]} = \{[x_1;\ x_2;\ \ldots;\ x_n] | x_i \in \Sigma^* \quad \text{for } 1 \leq i \leq n\} \cup \{[\ ]\}$$

$$\underline{\Sigma}^{[+]} = \{[x_1;\ x_2;\ \ldots;\ x_n] | x_i \in \Sigma^* \quad \text{for } 1 \leq i \leq n\} = \underline{\Sigma}^{[*]} - \{[\ ]\}$$

are called the *universal* list language and the [ ]-*free* (null-free) *universal* list language, respectively, over the alphabet Σ; the superscripts $^{[*]}$ and $^{[+]}$ stay for any number of components, including none, and at least one component, respectively.

A list $\underline{x}$ is said to be $[\varepsilon]$-*free* (epsilon-free) if and only if it does not contain any ε-component (empty component), *i.e.*, iff $\underline{x} = [x_1; x_2; \ldots; x_n]$ and $x_i \in \Sigma^+$ for any n and i with $1 \leq i \leq n$; the null list [ ] is conventionally considered as being $[\varepsilon]$-free (for it contains no component at all). The concept of [ ]-freedom must not be confused with that of $[\varepsilon]$-freedom. The ε-components can be omitted in a list, provided that the sourrounding separators $(;)$ are preserved; for instance $[a; \varepsilon] = [a;]$, $[\varepsilon; a] = [;a]$ and $[\varepsilon; \varepsilon] = [;]$.

A *sublist* of a list $\underline{x} \in \underline{\Sigma}^{[*]}$ is a subsequence of the components of $\underline{x}$, *i.e.* if $\underline{x} = [x_1; x_2; \ldots; x_n]$, for some $n \geq 1$, then $\underline{x}' = [x_k; x_{k+1}; \ldots; x_h]$, for some $1 \leq k \leq h \leq n$, is a sublist of $\underline{x}$. The null list is conventionally assumed to be a sublist of any other list, including itself. The sublist $\underline{x}'$ is *proper* if and only if $\underline{x}' \neq \underline{x}$. It is evident that a list is $[\varepsilon]$-free if and only if it admits no sublist of the type $[\varepsilon; \ldots; \varepsilon]$.

2.2. Basic operations on lists

We define two operations on lists, namely *list merge* and *list catenation*, and their closures, starting from merge. In this section we shall make use of two lists $\underline{x} = [x_1; x_2; \ldots; x_m]$ and $\underline{y} = [y_1; y_2; \ldots; y_n]$, with $m, n \geq 1$, over some alphabet Σ.

DEFINITION 2.1 (List Merge): $\| : \underline{\Sigma}^{[*]} \times \underline{\Sigma}^{[*]} \to \underline{\Sigma}^{[*]}$ *is a binary operation on lists, named list merge, acting through the mapping:*

$$[x_1; x_2; \ldots; x_m] \| [y_1; y_2; \ldots; y_n] \mapsto [x_1\, y_1; x_2\, y_2; \ldots; x_m\, y_m; y_{m+1}; \ldots; y_n] \qquad m < n$$

$$[x_1; x_2; \ldots; x_m] \| [y_1; y_2; \ldots; y_n] \mapsto [x_1\, y_1; x_2\, y_2; \ldots; x_m\, y_m] \qquad m = n$$

$$[x_1; x_2; \ldots; x_m] \| [y_1; y_2; \ldots; y_n] \mapsto [x_1\, y_1; x_2\, y_2; \ldots; x_n\, y_n; y_{n+1}; \ldots; x_m] \qquad m > n$$

and posing [ ]$\|\underline{x} = \underline{x}\|$[ ] $= \underline{x}$.

Merge is always written in infix notation. From now on, we shall call list merge simply as merge, as there is no danger of making confusion with

strings. Note that merge is associative. Moreover, it holds $\underline{x}\|[\varepsilon] = [\varepsilon]\|\underline{x} = \underline{x}$, for any list $\underline{x} \neq [\ ]$, and $\underline{x}\|\underline{x} = \underline{x}$ whenever the list $\underline{x} = [x_1; x_2; \ldots; x_n]$ is such that $x_i = \varepsilon$ for every $1 \leq i \leq n$.

A merge is associative, it is possible to define its closure;

DEFINITION 2.2 (List Merge Closure): $^{\|*} : \underline{\Sigma}^{[*]} \to \wp(\underline{\Sigma}^{[*]})$ *is a unary operation on lists, the closure of list merge, s.t.* $\underline{x}^{\|*} = \bigcup_{i=0}^{\infty} \underline{x}^{\|i}$, where $\underline{x}^{\|i} = \underbrace{\underline{x}\|\underline{x}\|\ldots\|\underline{x}}_{i \text{ times}}$, *for* $i \geq 1$, *and posing* $\underline{x}^{\|0} = [\ ]$.

Note that $[\ ]^{\|*} = \{[\ ]\}$ and that merge closure is idempotent, *i.e.* $(\underline{x}^{\|*})^{\|*} = \underline{x}^{\|*}$, for any list $\underline{x} \in \underline{\Sigma}^{[*]}$. The superscript $^{\|+}$ denotes the $[\ ]$-free closure of list merge.

DEFINITION 2.3 (List Catenation): $\bullet : \underline{\Sigma}^{[*]} \times \underline{\Sigma}^{[*]} \to \underline{\Sigma}^{[*]}$ *is a binary operation on lists, named list catenation, acting through the mapping:*

$$[x_1; x_2; \ldots; x_m] \bullet [y_1; y_2; \ldots; y_n] \mapsto [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]$$

and posing $[\ ] \bullet \underline{x} = \underline{x} \bullet [\ ] = \underline{x}$.

Catenation is always written in infix notation. A separator $(;)$ is always inserted between the two lists, but if one is the null list $[\ ]$. Note that list catenation is associative and that in general $[\varepsilon] \bullet \underline{x} \neq \underline{x} \bullet [\varepsilon] \neq \underline{x}$ as, for instance, $[\varepsilon]\bullet[a; b; c] = [\varepsilon; a; b; c] = [; a; b; c]$ and $[a; b; c]\bullet[\varepsilon] = [a; b; c; \varepsilon] = [a; b; c;]$. Briefly, the list $[\varepsilon]$ is not a neutral element for catenation.

A list catenation is associative, it is possible to define its closure:

DEFINITION 2.4 (List Catenation Closure: $^* : \underline{\Sigma}^{[*]} \to \wp(\underline{\Sigma}^{[*]})$ *is a unary operation on lists, the closure of list catenation, s.t.* $\underline{x}^* = \bigcup_{i=0}^{\infty} \underline{x}^i$, *where* $\underline{x}^i = \underbrace{\underline{x} \bullet \underline{x} \bullet \ldots \bullet \underline{x}}_{i \text{ times}}$, *for* $i \geq 1$, *and posing* $\underline{x}^0 = [\ ]$.

Note that $[\ ]^* = \{[\ ]\}$, that $[\varepsilon]^* = \{[\varepsilon]^k | k \geq 0\} \supset \{[\varepsilon]\}$ and that catenation closure is idempotent, *i.e.* $(\underline{x}^*)^* = \underline{x}^*$, for any list $\underline{x} \in \underline{\Sigma}^{[*]}$. The superscript $^+$ denotes the $[\ ]$-free closure of list catenation.

List catenation closure and $[\ ]$-free list catenation closure are denoted through the superscript * and $^+$, like the Kleene star and cross for strings, whereupon merge closure and $[\ ]$-free merge closure are denoted through the new superscripts $^{\|*}$ and $^{\|+}$, which are mnemonic for the merge operator.

From now on, list catenation will be no longer explicitly indicated by $\bullet$ (we use the traditional multiplicative notation), but in some exceptional case.

We shall refer to list catenation simply as catenation; from the context the type of arguments (list or string) of catenation will always be clear.

Note that neither merge is distributive w.r.t. catenation nor catenation is distributive w.r.t. merge, in general. Merge, catenation and their closures extend to sets of lists in the natural way. Clearly, in this extended sense both merge and catenation are distributive w.r.t. union.

2.3. Auxiliary operations on lists

Lists are similar to strings, yet some operations which are natural on strings are not well-defined or have multiple distinct interpretations on lists, such as for instance permutation, reflection, substitution, etc. Moreover, the traditional families of languages are defined over strings, hence comparisons with list families are not immediate. We introduce a *flattening* function to allow for this comparison:

DEFINITION 2.5 (Flattening): *The function* $\backslash\backslash : \underline{\Sigma}^{[*]} \to \Sigma^*$ *on lists onto strings is acting through the mapping:*

$$\backslash[x_1;\ x_2;\ \ldots;\ x_n]\backslash \mapsto x_1\, x_2\, \ldots\, x_n \qquad \text{for} \quad \underline{x} \in \underline{\Sigma}^{[*]}$$

where $\underline{x} = [x_1;\ x_2;\ \ldots;\ x_n]$ *for* $n \geq 1$, *and posing* $\backslash[\ ]\backslash = \varepsilon$.

Flattening is always denoted by enclosing in backslashes. An example is: $\backslash[a;\ bc;\ \varepsilon;\ d;\ \varepsilon]\backslash = abcd$. The inverse of flattening is called *free segmentation* operator, as it makes lists out of strings by freely segmenting them into components, *i.e.* $\backslash x\backslash^{-1}$, where $x \in \Sigma^*$ is a string, is the set of all the lists $\underline{x} \in \underline{\Sigma}^{[*]}$ having as flattened image precisely the string x.

In subsequent sections it will be necessary to preserve the separator $(;)$ also in the strings. Define an alphabet extended with the new character $(;)$ as $\Sigma_S = \Sigma \oplus \{;\}$, where Σ is any ordinary alphabet; the index $_S$ stays for "separator", *i.e.* the character $(;)$. It is convenient to introduce the following two conversion functions:

DEFINITION 2.6 (Conversion): *Define the following conversion functions:*

$$\mathit{list} : \Sigma_S^* \to \underline{\Sigma}^{[+]} \quad \mathit{list}\,(x_1; x_2; \ldots; x_n) \mapsto [x_1; x_2; \ldots; x_n] \quad \mathit{list}\,(\varepsilon) \mapsto [\varepsilon]$$

$$\mathit{string} : \underline{\Sigma}^{[+]} \mapsto \Sigma_S^* \quad \mathit{string}\,([x_1;\ x_2;\ \ldots;\ x_n]) \mapsto x_1;\ x_2;\ \ldots;\ x_n$$

$$\mathit{string}\,([\varepsilon]) \mapsto \varepsilon$$

where $x_i \in \Sigma^*$, *for any* n and i *with* $1 \leq i \leq n$.

The two functions *list* and *string* transform a string into a list and a list into a string, respectively, preserving the structure induced by the presence of the separator $(;)$. Examples are: $list\,(a;\ bc;\ ;\ d;) = [a;\ bc;\ \varepsilon;\ d;\ \varepsilon]$ and $string\,([\varepsilon;\ abc;\ \varepsilon;\ d]) =; abc;\ ;\ d$, respectively.

Note that flattening, instead, removes the list substructure from its list argument. Given the set of lists (resp. words) $\underline{L} \subseteq \underline{\Sigma}^{[+]}$ (resp. $L \subseteq \Sigma_S^*$), the set $string\,(\underline{L})$ (resp. $list\,(L)$) is the $word$ (resp. $list$) image *associated* with the $list$ (resp. $word$) set $\underline{L}$ (resp. L). It is fairly evident that the conversion functions *list* and *string* are one-to-one.

We shall refer to the functions *list* and *string* as the *segmentation* of a string into a list and the *compaction* of a list into a string, respectively. Note that it is impossible to obtain the null list [ ] by segmenting a string: in fact, segmenting a non-empty string $x \neq \varepsilon$ clearly cannot yield the null list [ ] and segmenting the empty string ε yields the list $[\varepsilon]$, which also differs from the null list [ ]. Therefore the functions *list* and *string* are left undefined as for the null list [ ].

STATEMENT 2.1 (Dependence): *Given the natural projection* $\pi : \Sigma_S^* \to \Sigma^*$ *cancelling the separator* $(;)$, *the following functional relationships hold:*

$$(string \circ list)\,(x) = x \qquad (list \circ string)\,(\underline{x}) = \underline{x}$$
$$(\pi \circ string)\,(\underline{x}) = \backslash\underline{x}\backslash \qquad (list \circ \pi^{-1})\,(x) = \backslash x\backslash^{-1}$$

for any string $x \in \Sigma^*$ *and any list* $\underline{x} \in \underline{\Sigma}^{[+]}$.

Proof: It follows from that the functions *list* and *string* are one-to-one and both flattening and the projection π cancel the separator $(;)$. □

The statement implies that flattening $(\backslash\backslash)$, segmentation (*list*), compaction (*string*) and free-segmentation $(\backslash\backslash^{-1})$ are not functionally independent.

3. LIST LANGUAGES

A *list language* is any subset of the universal list language $\underline{\Sigma}^{[*]}$; a *list family* is a set of list languages. A list language not containing the null list [ ] is said to be [ ]-free; a family of [ ]-free list languages is also said to be [ ]-free; similarly for $[\varepsilon]$-freedom.

3.1. Fair expressions

We define a finitary generative model for some list languages, which extends the well-known finitary generative model of regular expressions,

by using finite lists as generators and allowing merge and catenation, plus their closures and union, as basic operators; this new model is named *fair expression.*

DEFINITION 3.1 (Fair Expression): *A Fair Expression, briefly a FE, over the alphabet* Σ *is one of the following:*

(1) *Any finite list* $\underline{x}$ *(including the null list* [ ]*) over the alphabet* Σ *is a FE.*

(2) *If* f, f_1 *and* f_2 *are FE's over the alphabet* Σ, *then:*

(a) $f_1 \cup f_2$ *is a FE.*

(b) $f_1 \| f_2$ *is a FE.*

(c) $f^{\|*}$ *is a FE (and also* $f^{\|+}$*).*

(d) $f_1 \bullet f_2$ *is a FE (also written multiplicatively,* $f_1 f_2$*).*

(e) f^* *is a FE (and also* f^+*).*

Nothing else is a fair expression, but what is obtained by a finite number of recursive applications of the rule (1) *and rules* (2).

The finite lists appearing inside a FE are its *generators.* A FE is said to be $[\varepsilon]$-*free* if and only if its generators are $[\varepsilon]$-free lists. The property of $[\varepsilon]$-freedom of a FE is obviously decidable.

STATEMENT 3.1 (Comparison): *Fair expressions contain two restrictions which are isomorphic to regular expressions, namely:*

• *Fair expressions restricted to contain union, merge and merge closure* (but not catenation and its closure), *and with one-component generators,* i.e. *of the type* $[x]$, *for some string* $x \in \Sigma^*$.

• *Fair expressions restricted to contain union, catenation and catenation closure* (but not merge and its closure), *and with generators containing one-letter non-empty components,* i.e. *of the type* $[x_1; \ldots; x_n]$, *for some characters* $x_i \in \Sigma$ *with* $1 \leq i \leq n$.

The isomorphism is given by the flattening function (see definition 2.5).

Proof: It suffices to check that in the two cases the operations are preserved

by flattening and that flattening is one-to-one.

$$\backslash[x]\|[y]\backslash = \backslash[xy]\backslash = x \cdot y$$
$$\backslash[x]\backslash = \backslash[y]\backslash \quad \Rightarrow \quad x = y \quad \Rightarrow \quad [x] = [y]$$
$$\backslash[x_1; \ldots; x_m] \bullet [y_1; \ldots; y_n]\backslash = \backslash[x_1; \ldots; x_m; y_1; \ldots; y_n]\backslash$$
$$= x_1 \ldots x_m\, y_1 \ldots y_n = \backslash[x_1; \ldots; x_m]\backslash \cdot \backslash[y_1; \ldots; y_n]\backslash$$
$$\backslash[x_1; \ldots; x_m]\backslash = \backslash[y_1; \ldots; y_n]\backslash \quad \Rightarrow \quad m = n$$
$$\text{and} \quad x_1 = y_1 \ldots x_n = y_n \quad \Rightarrow \quad [x_1; \ldots; x_m] = [y_1; \ldots; y_n]$$

for the strings $x, y \in \Sigma^*$ and the characters $x_i, y_j \in \Sigma$, with $1 \leq i, j \leq m, n$. This shows preservation and injectivity; the surjectivity of flattening is evident, whence bijectivity. □

3.2. Fair languages

Any FE defines, in the obvious way, a possibly infinite set of finite lists. The notation $L(f)$ indicates the list language generated by the FE f. In order to simplify the parenthetizations, we shall assume that the closures take precedence over any other operator and that both merge and catenation take precedence over union. The precedence between merge and catenation must be indicated explicitly by means of parentheses.

DEFINITION 3.2 (Fair List Language): *A list language $\underline{L}$ is a Fair List Language – denoted as FLL – if and only if it is generated by some FE f,* i.e. *iff $\underline{L} = L(f)$. A word language L is a flattened Fair List Language – denoted as fFLL – if and only if it is the flattening of a Fair List Language,* i.e. *iff $L = \backslash L(f)\backslash$.*

Flattening is not a basic operation of the FE model; it is only applied at the end, when we want to pass from a FLL to a fFLL, which is directly comparable to a word language. A FLL is said to be $[\varepsilon]$-*free* if and only if it contains only $[\varepsilon]$-free lists and is said to be [ ]-free if and only if it does not contain the null list [ ]. It is immediate to see that the properties of $[\varepsilon]$ and [ ]-freedom are independent in FLL; examples of the four possible cases are easy to construct.

Note than an $[\varepsilon]$-free FE generates an $[\varepsilon]$-free FLL, but a non-$[\varepsilon]$-free FE may well generate an $[\varepsilon]$-free FLL. For instance, the non $[\varepsilon]$-free FE $f = [a; \varepsilon]\|[\varepsilon; b]$ generates the (finite) FLL $L(f) = \{[a; b]\}$, which is $[\varepsilon]$-free.

3.3. Regular list languages

In order to perform a more systematic comparison of FLL and fFLL with traditonal families (e.g. the Chomsky hierarchy) and to set up some tools for transforming at least the simplest FE's and FLL's, it is useful to have a notion of regularity in FLL.

Simple tricks, such as saying that regular list languages are defined by FE's only using catenation or merge, prove immediately to be too naive. In the former case, they coincide with the regular languages over the free monoid Σ^* (here is the simplicity), yet the list components would consist of strings of fixed length (here is the naiveness). In the latter case they coincide with the regular languages over the finite direct product of free monoids Σ^* [Brs79] (here again is the simplicity), yet the lists would have an upper bounded number of components (here again is the naiveness). It is clear that in both cases the separators $(;)$ are placed in a trivial way. Moreover, the null list $[\ ]$ is somewhat disturbing.

In the following a more general proposal of regular list language, based on the notion of flattening, is exposed. Assume that Σ is an alphabet and that $\Sigma_S = \Sigma \cup \{;\}$ is the same alphabet as Σ, but extended with the separator $(;)$.

DEFINITION 3.3 (Regular List Language): *A $[\ ]$-free list language $\underline{R} \subseteq \underline{\Sigma}^{[+]}$ is a Regular List Language – denoted as RLL – if and only if there exists a regular language $R \subseteq \Sigma_S^*$ s.t. $\underline{R} = list(R)$.*

The above definition states that a RLL is a list language that is obtained from a string language by segmenting its strings into list components in some "regular" way, a natural extension to the list domain of the classical notion of regular language. Therefore RLL's are always assumed to be $[\ ]$-free (*see* also section 2.3); however, RLL's may well contain non-$[\varepsilon]$-free lists.

There exists a smaller, but important, subfamily of RLL, based on the notion of free segmentation of strings:

DEFINITION 3.4 (simple Regular List Language): *A $[\ ]$-free list language $\underline{R} \subseteq \underline{\Sigma}^{[+]}$ is a simple Regular List Language – denoted as sRLL – if and only if there exists a regular language $R \subseteq \Sigma^*$ s.t. $\underline{R} = \backslash R \backslash^{-1} - \{[\ ]\}$.*

Note that $sRLL = (list \circ \pi^{-1})\ (\mathcal{R})$ or $\pi^{-1}\ (\mathcal{R}) = string(sRLL)$, where $\pi : \Sigma_S^* \to \Sigma^*$ is the natural projection cancelling the separator $(;)$ and $\mathcal{R}$ is the family of regular (word) languages. A sRLL differs from a RLL in that the lists of a sRLL exhibit a free distribution of separators $(;)$; also the sRLL's are always $[\ ]$-free, but may contain non-$[\varepsilon]$-free lists. In the sequel

we shall shorten the writing $\backslash R\backslash^{-1} - \{[\ ]\}$ into $\backslash R\backslash^{-1}$, understanding that sRLL is always [ ]-free. In section 7.2 sRLL will be used for proving some non-trivial results.

4. SIMPLE EXAMPLES OF LIST LANGUAGES

To study the various families of list languages introduced in section 3, the following notation is generally adopted:

Families of List Languages	
FLL, RLL, sRLL	Fair, Regular, simple Regular List Languages
Families of Word Languages	
fFLL	flattened Fair List Languages

Recall that a FLL is any list language generated by a fair expression, a RLL is any list language such that its word image still containing the separator $(;)$ is regular and a sRLL is any list language obtained by saturating some regular language with respect to the introduction of the separator $(;)$.

We start by investigating the families FLL and fFLL. Some examples will help in showing the generative power of FE. Sometimes we number the parentheses nests in the displayed FE's (on the right side), as a visual aide to find the matching between the FE's and the proposed instances thereof.

Example 4.1 ($\{a^n\, b^n\, c^n\}$): The language $L = \{a^n\, b^n\, c^n | n \geq 0\}$ is a fFLL. In fact, $L = \backslash \underline{L} \backslash$, where $\underline{L}$ is a FLL generated by the $[\varepsilon]$-free FE f_1:

$$f_1 = [a;\ b;\ c]^{\|*}$$

where the alphabet is $\Sigma = \{a,\ b,\ c\}$ and the unique generator is $[a;\ b;\ c]$. The example generalizes easily to any alphabet cardinality. □

One has, for instance:

$$a^3\, b^3\, c^3 = \backslash [a^3;\ b^3;\ c^3] \backslash = \backslash [a;\ b;\ c] \| [a;\ b;\ c] \| [a;\ b;\ c] \backslash \in \backslash L\,(f_1) \backslash$$

Example 4.2 (Commutation): The closures $*$ and $^{\|*}$ do not commute. Given the two $[\varepsilon]$-free FE's f_2 and f_3, it holds:

$$\begin{aligned} f_2 &= ([a;\ b;\ c]^*)^{\|*} \\ &= \{[a^{n_1};\ b^{n_1};\ c^{n_1};\ a^{n_2};\ b^{n_2};\ c^{n_2};\ \ldots] | n_i \geq n_{i+1} \geq 0 \text{ for } i \geq 1\} \cup \{[\,]\} \\ f_3 &= ([a;\ b;\ c]^{\|*})^* \\ &= \{[a^{n_1};\ b^{n_1};\ c^{n_1};\ a^{n_2};\ b^{n_2};\ c^{n_2};\ \ldots] | n_i \geq 0 \text{ for } i \geq 1\} \cup \{[\,]\} \end{aligned}$$

where the alphabet is $\Sigma = \{a, b, c\}$ and in both cases the unique generator is $[a; b; c]$. The example generalizes easily to any alphabet cardinality. □

Clearly one has $L(f_2) \subseteq L(f_3)$, since:

$$[a^2; b^2; c^2; a; b; c] = ([a; b; c]\,[a; b; c])\|[a; b; c] \in L(f_2)$$
$$[a^2; b^2; c^2; a; b; c] = ([a; b; c]\|\,[a; b; c])\,[a; b; c] \in L(f_3)$$

However, it is easy to see that:

$$[a; b; c; a^2; b^2; c^2] = [a; b; c]\,([a; b; c]\|[a; b; c]) \in L(f_3) - L(f_2)$$

because any instance of f_2 first concatenates the generators $[a; b; c]$ and merges the obtained strings only later, which implies that in no list of $L(f_2)$ the sublist $[d^m; \ldots; d^n]$, with $m < n$ and $d = a, b, c$, can occur. Hence one proves $L(f_2) \subset L(f_3)$, *i.e.* strict inclusion.

Example 4.3 (Anagram): The language of the anagrams of $\{a^n\, b^n\, c^n | n \geq 1\}$, from now on denoted *Anagrams of* $\{a^n\, b^n\, c^n | n \geq 1\}$, is a fFLL, generated by the non- $[\varepsilon]$-free FE's f_4 or f_5:

$$f_4 = (([\varepsilon]^*\,[a])_2\|([\varepsilon]^*\,[b])_3\|([\varepsilon]^*\,[c])_4)_1^{\|+}$$
$$f_5 = \underbrace{([\varepsilon]^*\,[a]\,[\varepsilon]^*\,[b]\,[\varepsilon]^*\,[c] \cup [\varepsilon]^*\,[b]\,[\varepsilon]^*\,[a]\,[\varepsilon]^*\,[c] \cup \ldots)^{\|+}}_{\text{all the permutations of } a, b, c \text{ that leave the } \varepsilon\text{'s fixed}}$$

where the alphabet is $\Sigma = \{a, b, c\}$ and in both cases the generators are $[\varepsilon]$, $[a]$, $[b]$ and $[c]$. The FE's f_4 and f_5 generalize immediately to any alphabet cardinality. □

Consider, for instance, how the following string is generated using f_4 or f_5, respectively:

$$\begin{aligned} abbcaacccbba &= \backslash[a; b; b; c; a; a; c; c; c; b; b; a]\backslash \\ &= \backslash(([a])_2\|([\varepsilon]\,[b])_3\|([\varepsilon]^3\,[c])_4)_1\| \\ &\quad \|(([\varepsilon]^4\,[a])_2\|([\varepsilon]^2\,[b])_3\|([\varepsilon]^6\,[c])_4)_1\| \\ &\quad \|(([\varepsilon]^5\,[a])_2\|([\varepsilon]^9\,[b])_3\|([\varepsilon]^7\,[c])_4)_1\| \\ &\quad \|(([\varepsilon]^{11}\,[a])_2\|([\varepsilon]^{10}\,[b])_3\|([\varepsilon]^8\,[c])_4)_1\backslash \\ &\in \backslash L(f_4)\backslash \\ abbcaacccbba &= \backslash[a; b; b; c; a; a; c; c; c; b; b; a]\backslash \\ &= \backslash([a]\,[b]\,[\varepsilon]\,[c])\|([\varepsilon]^2\,[b]\,[\varepsilon]\,[a]\,[\varepsilon]\,[c])\| \\ &\quad \|([\varepsilon]^5\,[a]\,[\varepsilon]\,[c]\,[\varepsilon]\,[b])\|([\varepsilon]^8\,[c]\,[\varepsilon]^7\,[b]\,[a])\backslash \\ &\in \backslash L(f_5)\backslash \end{aligned}$$

The FE f_4 generates the final list by producing intermediate lists having only a single non-empty one-letter component placed rightmost, then merging the intermediate list three by three – so that in any group of three the letters a, b and c occur in some order – and finally computing the merge closure of the groups. The FE f_5 generates the final list by merging intermediate lists, each one containing exactly three letters a, b and c – possibly permuted – placed in distinct components; the remaining components are empty strings. Such lists are computed by reading the string left to right, collecting all the triples of distinct letters. The ε-components cause the letters to be freely shifted rightwards. Note that both generation mechanisms are heavily non-deterministic. By using the closure $^{\|*}$ instead of $^{\|+}$ the language *Anagrams* of $(\{a^n\, b^n\, c^n | n \geq 0\})$ is obtained.

Example 4.4 (Universal): The universal list language $\Sigma^{[*]}$, over the alphabet $\Sigma = \{a, b\}$, is a FLL, generated by the non-$[\varepsilon]$-free FE's f_6 or f_7:

$$f_6 = (([\varepsilon] \cup [a] \cup [b])_2^{\|*})_1^* \qquad \text{or} \qquad f_7 = (([\varepsilon] \cup [a] \cup [b])_2^*)_1^{\|*}$$

where in both cases the generators are $[\varepsilon]$, $[a]$ and $[b]$. The FE's f_6 and f_7 generalize immediately to any alphabet cardinality. Using the $[\ ]$-free closures the universal $[\ ]$-free list language is obtained. □

One has, for instance, using the FE's f_6 and f_7:

$$[abaa; b; \varepsilon; ab] = (([a])_2 \| ([b])_2 \| ([a])_2 \| ([a])_2)_1 (([b])_2)_1 (([\varepsilon])_2)_1 (([a])_2 \| ([b])_2)_1 \in L(f_6)$$

$$\begin{aligned}[abaa; b; \varepsilon; ab] = {} & (([a])_2)_1 \| (([b])_2)_1 \| (([a])_2)_1 \| (([a])_2)_1 \| \\ & \| (([\varepsilon])_2\ ([b])_2)_1 \| (([\varepsilon])_2\ ([\varepsilon])_2\ ([\varepsilon])_2)_1 \| \\ & \| (([\varepsilon])_2\ ([\varepsilon])_2\ ([\varepsilon])_2\ ([a])_2)_1 \| (([\varepsilon])_2\ ([\varepsilon])_2\ ([\varepsilon])_2\ ([b])_2)_1 \\ & \in L(f_7)\end{aligned}$$

Note that $(\{[a], [b]\}^{\|*})^*$ is not the universal list language, as it contains all and only $[\varepsilon]$-free lists.

This example also shows that in some cases the closures * and $^{\|*}$ may commute.

Example 4.5 (Pattern Language); The Pattern Languages of Angluin [An80] are in fFLL. For instance, given the pattern $p = aX\, bY\, cX\, dY$ – containing the variables X, Y and the constants a, b, c - the Pattern Language generated by p, substituting to the variables X and Y strings over the alphabet $\{e, f\}$, is:

$$p(e, f) = \{aubvcudv | u, v \in \{e, f\}^*\}$$

and is generated by the flattened FE:

$$f_8 = [a;\ \varepsilon;\ b;\ \varepsilon;\ c;\ \varepsilon;\ d;\ \varepsilon]\|\|$$
$$\|([\varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ ] \cup [\varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon] \cup$$
$$\cup [\varepsilon;\ \varepsilon;\ \varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ e] \cup [\varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f])^{\|*}$$

where the alphabet is $\Sigma = \{a,\ b,\ c,\ d,\ e,\ f\}$ and the generators are $[a;\ \varepsilon;\ b;\ \varepsilon;\ c;\ \varepsilon;\ d;\ \varepsilon]$, $[\varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ ]$, ..., $[\varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f]$. □

One has, for instance, using the FE f_8:

$$aef\,fbfefcef\,fdfef = \backslash[a;\ eff;\ b;\ fef;\ c;\ eff;\ d;\ fef]\backslash$$
$$= [a;\ \varepsilon;\ b;\ \varepsilon;\ c;\ \varepsilon;\ d;\ \varepsilon]\|\|([\varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ e;\ \varepsilon;\ \varepsilon\ ]\|\|$$
$$\|[\varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon]\|\|[\varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon]\|\|$$
$$\|[\varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f]\|\|[\varepsilon;\ \varepsilon;\ \varepsilon;\ e;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ e]\|\|$$
$$\|[\varepsilon;\ \varepsilon;\ \varepsilon;\ f;\ \varepsilon;\ \varepsilon;\ \varepsilon;\ f])\backslash \in \backslash L(f_8)\backslash$$

The construction here shown for finding the FE equivalent to a given pattern is general with respect to Σ and p. The inclusion of Pattern Languages in fFLL is strict; Pattern Languages are closed with respect to the intersection with regular languages [Ang80], whereas it will be proved in statement 7.6 that the family fFLL is not.

In general, also the rational closure of the family of Pattern Languages is (strictly) included in the family of fFLL; for instance, the closure language $p.(e,\ f)^*$ is generated by the flattened FE f_8^*.

These five examples have hopefully given the reader some intuitive understanding of the inner structure of FLL, though not yet in a systematic way. None of them is RLL, but example 4.4, which actually is sRLL (in the [ ]-free version), and hence also RLL, because its associated word language is the free monoid Σ^* with words saturated with respect to the introduction of the separator $(;)$. As for a direct example of a sRLL with a non-trivial word image and of a RLL which is not sRLL, consider the following FE's:

Example 4.6 (Regular List Language): The following FE's generate a sRLL and a RLL, respectively:

$$f_9 = ([\varepsilon]^*\,([ab]^{\|+} \cup ([a]\|([\varepsilon]^*\,[ba]^{\|*})_4)_3\,[\varepsilon]^*\,([b]\|([\varepsilon]^*\,[ab]^{\|*})_6)_5)_2\,[\varepsilon]^*)_1^+$$
$$\in \text{sRLL}$$
$$f_{10} = ([\varepsilon]^*\,[ab]^{\|+}\,[\varepsilon]^*)^+ \in \text{RLL}$$

where in both cases the alphabet is $\Sigma = \{a,\ b\}$, the generators of f_9 are $[\varepsilon]$, $[a]$, $[b]$, $[ab]$ and $[ba]$, and the generators of f_{10} are $[\varepsilon]$ and $[ab]$. □

Both the list languages $L(f_9)$ and $L(f_{10})$ have the same flattened image, which is the (non-trivial) regular word language $(ab)^+$. The list languages generated by such FE's differ as follows, respectively:

$$[s_1;\ s_2;\ \ldots;\ s_n] \in L(f_9) \quad s_i \in \varepsilon \cup (ab)^+ \cup a(ab)^* \cup b(ab)^* \quad 1 \leq i \leq n$$
$$[s_1;\ s_2;\ \ldots;\ s_n] \in L(f_{10}) \quad s_i \in \varepsilon \cup (ab)^+ \quad 1 \leq i \leq n$$

The list language $L(f_9)$ is sRLL because it is the saturation of the regular word language $(ab)^+$ with respect to the introduction of the separator $(;)$. The FE f_9 first generates in all possible ways intermediate lists with an arbitrary number of components, but where either one or two components are non-empty. If only one component is non-empty, then it is $(ab)^+$, otherwise the two non-empty components (which can be separated by empty components) are orderly $a(ba)^*$ and $b(ab)^*$. The intermediate lists are then concatenated freely once or more times. For instance:

$$[\varepsilon;\ ab;\ a;\ \varepsilon;\ ba;\ \varepsilon;\ b]$$
$$= ([\varepsilon]\,([ab])_2\,[\ ])_1\,([\ ]\,(([a]\|([\varepsilon]^2\,[ba])_4)_3\,[\varepsilon]\,([b]\|([\ ]\,[\ ])_6)_5)_2\,[\ ])_1$$

Clearly the final lists have an arbitrary number of components, which can be the empty string ε or any substring of $(ab)^+$, but the concatenation of the non-empty components (at least one) always yields a string of the type $(ab)^+$.

The list language $L(f_{10})$ is RLL because its word image $string(L(f_{10})) = (;)^*\,(ab)^+\,(;\,(ab)^*)^*$ is regular. The FE f_{10} generates intermediate lists with an arbitrary number of components, but where exactly one component is non-empty, and is $(ab)^+$. The intermediate lists are then concatenated freely once or more times. For instance:

$$[\varepsilon;\ ab;\ \varepsilon;\ \varepsilon;\ abab;\ \varepsilon] = ([\varepsilon]\,[ab]\,[\ ])\,([\varepsilon]^2\,[ab]^{\|2}\,[\varepsilon])$$

Clearly the final lists have an arbitrary number of components, but each non-empty component (at least one) is always of the type $(ab)^+$. The list language $L(f_{10})$ is not sRLL, because the word language $string(L(f_{10}))$ is not the saturation of the regular word language $(ab)^+$ with respect to the introduction of the separator $(;)$.

5. A MORE COMPLEX EXAMPLE

The next example is important and requires a more extensive treatment. This example will be reused later to prove a significant non-closure

property. It is concerned with the so-called AntiDyck language by Franchi Zannettacci-Vauquelin [Fra80] and Brandenburg [Bra88].

DEFINITION 5.1 (AntiDyck Equivalence): *Let Σ be an alphabet, called natural alphabet, let $\overline{\Sigma}$ be a disjoint copy of Σ, called barred alphabet, and pose $\Delta = \Sigma \cup \overline{\Sigma}$. Define the binary relation $E_0 \subseteq \Delta^2$ over the strings so that:*

$$x\, E_0\, y \quad \Leftrightarrow \quad x = au\overline{a}v \qquad \text{and} \qquad y = uv$$
$$\text{where} \quad x,\, y \in \Delta^*,\, u \in \Sigma^* \quad \text{and} \quad v \in \Delta^*$$

Let the binary equivalence relation over the strings $E = (E_0 \cup E_0^{-1})^* \subseteq \Delta^2$ be defined as the reflexive, symmetric and transitive closure of the binary relation E_0. The binary relation E is called the AntiDyck equivalence.

For instance, it holds $ab\overline{a}\overline{b} =_E au\overline{a}v =_E b\overline{b}$, where $u = b$ and $v = \overline{b}$, but $ab\overline{b}\overline{a} \neq_E b\overline{b}$, because the reduction of $ab\overline{b}\overline{a}$ to $b\overline{b}$ would require that $u = b\overline{b}$, but $u \notin \Sigma^*$.

DEFINITION 5.2 (AntiDyck Language): *The AntiDyck language* [Fra80], *over the alphabet $\Delta = \Sigma \cup \overline{\Sigma}$, is defined as the set of all the strings of Δ^* that are equivalent to the empty string ε through the AntiDyck equivalence relation E. The language is denoted AntiDyck $(\Sigma) \subseteq \Delta^*$. It is a representative queue language (Franchi Zannettacci-Vauquelin* [Fra80], *and is also called FIFO language (Brandenburg* [Bra88]*).*

Here are some AntiDyck strings, for the natural alphabet $\Sigma = \{a,\, b\}$

$$\varepsilon,\ a\overline{a},\ ab\overline{a}\overline{b},\ aba\overline{a}b\overline{b}\overline{a}\overline{b},\ ab\overline{a}bb\overline{b}\overline{b}ab\overline{a},\ \ldots$$

They are all equivalent to the empty string ε through the AntiDyck equivalence $=_E$.

To show the procedure to check the validity of an AntiDyck string, consider the following successful reduction to ε of the AntiDyck string $ab\overline{a}bb\overline{b}\overline{b}ab\overline{a}$:

$$\begin{array}{rlll}
xu\overline{x}v =_E uv & u \in \Sigma^* & v \in \Delta^* \text{ and } x,\, \overline{x} \in \Sigma,\, \overline{\Sigma} \\
\mathbf{a}b\mathbf{\overline{a}}bb\overline{b}\overline{b}ab\overline{a} =_E bbb\overline{b}\overline{b}ab\overline{a} & u = b & v = bb\overline{b}\overline{b}ab\overline{a} \\
\mathbf{b}bb\mathbf{\overline{b}}\overline{b}ab\overline{a} =_E bb\overline{b}ab\overline{a} & u = bb & v = \overline{b}ab\overline{a} \\
\mathbf{b}b\mathbf{\overline{b}}ab\overline{a} =_E bab\overline{a} & u = b & v = ab\overline{a} \\
\mathbf{b}a\mathbf{\overline{b}}\overline{a} =_E a\overline{a} & u = a & v = \overline{a} \\
\mathbf{a}\mathbf{\overline{a}} =_E \varepsilon & u = \varepsilon & v = \varepsilon
\end{array}$$

AntiDyck could also be defined as the set of all the strings of completely unnested bracket pairs; the natural (resp. barred) characters of Σ (resp. $\overline{\Sigma}$) are the "open" (resp. "closed") brackets [Bre93, Bre94, Fri94]. Figure 1a shows the parentheses structure of the above sample string; all the AntiDyck strings shown above are of this type. Finally, AntiDyck could be defined as the set of the activity traces of a FIFO job shop: imagine that the characters of Σ are requests sent by clients to the job shop and that the characters of $\overline{\Sigma}$ are the services dispatched by the job shop in response to the requests; the time of arrival of the requests is free and each request deserves exactly one service, specific for that request. The job shop is FIFO if and only if the services are dispatched after and in the same order as the requests [Bre91]. Figure 1b shows the scheduling of the above sample string; all the AntiDyck strings shown above are of this type.

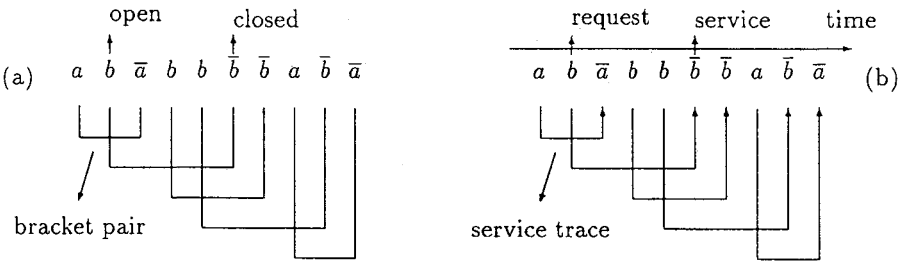

Figure 1. – Parentheses (a) and trace (b) interpretations of the AntiDyck language.

Example 5.1 (AntiDyck): AntiDyck is a fFLL obtained as the flattening of the FLL generated by the FE f_{11}:

$$f_{11} = ([\varepsilon]^* [a; \overline{a}] \cup [\varepsilon]^* [b; \overline{b}])^{\|*} \qquad \text{and} \qquad AntiDyck(\Sigma) = \backslash L(f_{11}) \backslash \subseteq \Delta^*$$

where $\Sigma = \{a, b\}$, $\Delta = \{a, b, \overline{a}, \overline{b}\}$ and the generators are $[\varepsilon]$, $[a; \overline{a}]$ and $[b; \overline{b}]$. The FE f_{11} generalizes immediately to any alphabet cardinality. We shall denote the list language $L(f_{11})$ as $\underline{AntiDyck}(\Sigma) \subseteq \underline{\Delta}^{[*]}$ and call it the list AntiDyck language. □

We justify the correctness of the FE f_{11} by working out a significant case. Referring to the valid AntiDyck string whose reduction to ε has been shown above, an example of expansion of f_{11} is the following:

$$\begin{aligned} a\overline{a}b\overline{b}\overline{b}a\overline{b}\overline{a} &= \backslash [ab;\ \overline{a}\overline{b}\overline{b};\ \overline{b}a\overline{b};\ \overline{a}] \backslash \\ &= \backslash ([a;\ \overline{a}]) \| ([\varepsilon;\ b;\ \overline{b}]) \| ([\varepsilon;\ \varepsilon;\ a;\ \overline{a}]) \| ([\varepsilon;\ b;\ \overline{b}]) \| ([b;\ \overline{b}]) \backslash \\ &\in \backslash AntiDyck(\{a, b\}) \backslash \end{aligned}$$

In general, to prove that the flattened image of any list generated by the FE f_{11} is a valid AntiDyck string, observe that f_{11} works by freely merging lists of the type: $[\varepsilon; \ldots; \varepsilon; a; \overline{a}]$ and $[\varepsilon; \ldots; \varepsilon; b; \overline{b}]$; we shall call them *generating* lists. The flattened images of the generating lists are the trivial AntiDyck strings $a\overline{a}$ and $b\overline{b}$. The generating lists are obtained by shifting to the right the generators $[a; \overline{a}]$ and $[b; \overline{b}]$ of f_{11} for an arbitrary number of ε-components. The merging of an arbitrary number of generating lists always yields a list whose flattened image is a valid AntiDyck string. In fact, the merging of the generating lists ensures that the following properties are preserved:

- A natural character, say a, is always followed by its barred copy $\overline{a}$ in the next list component, *i.e.* $[\ldots; \ldots a \ldots; \ldots \overline{a} \ldots; \ldots]$.
- If two natural characters occur in some order in the same component (but not necessarily adjacent), say $a \ldots b$, then their barred copies occur in the same order in the next list component, *i.e.* $\overline{a} \ldots \overline{b}$.

Therefore the FE f_{11} only generates lists whose flattened images are valid AntiDyck strings.

Conversely, to show that every AntiDyck string is a flattened image of some list generated by the FE f_{11}, we must first show how to segment an AntiDyck string into a list and then that such a list is really generated by f_{11}.

As for the segmentation step, proceed as follows. The AntiDyck string is scanned one-way from left to right by a read/write head, which reads the characters of the string and may insert, from time to time, the separator $(;)$, in the following way:

- A separator $(;)$ is inserted immediately to the left of the first barred character found (which certainly exists, unless the AntiDyck string is empty, which is a trivial case).
- Whenever the projection – over the barred alphabet $\overline{\Sigma}$ – of the string factor delimited between the last inserted separator $(;)$ (the rightmost one) and the current position of the read/write read coincides with the projection – over the natural alphabet Σ – of the string factor delimited between the last two inserted separators $(;)$ (the two rightmost ones), or between the left end of the string and the last inserted separator $(;)$ (the rightmost one) if the string still contains only one separator $(;)$, then the next separator $(;)$ is inserted immediately to the right of the current position of the read/write head, unless the head has reached the right end of the string.

The whole process is deterministic, so the result (an AntiDyck string segmented by separators $(;)$) is unique. The segmented AntiDyck string is then converted into a list (AntiDyck list) in the obvious way (use the

function *list*). The following table shows the segmentation process, applied to the same valid AntiDyck string as above.

Segmentation of the AntiDyck string										
AntiDyck string:	a	b	$\overline{a}$	b	b	$\overline{b}$	$\overline{b}$	a	$\overline{b}$	$\overline{a}$
Pass 1	a	b;								
Pass 2	a	b;	$\overline{a}$			$\overline{b}$;				
Pass 3		;		b	b	;	$\overline{b}$		$\overline{b}$;	
Pass 4		;				;		a	;	$\overline{a}$
Segmented string:	a	b;	$\overline{a}$	b	b	$\overline{b}$;	$\overline{b}$;	a	$\overline{b}$	$\overline{a}$
AntiDyck list:	$[ab$;		$\overline{a}bb\overline{b}$;				$\overline{b}a\overline{b}$;			$\overline{a}]$

After obtaining the AntiDyck list, we must extract from it the generating lists. This task is easy, as it suffices to scan the AntiDyck list one-way from left to right and collect the letters pairs; $\ldots a \ldots; \ldots \overline{a} \ldots$, etc., which correspond naturally to the generators of f_{11}, e.g. $[a; \overline{a}]$, etc., and then build the generating lists by prepending to them as many ε-components as it is required by their position in the AntiDyck list. The following table shows the extraction process

Extraction of the generating lists				
Components:	1	2	3	4
AntiDyck list:	$[ab$;	$\overline{a}bb\overline{b}$;	$\overline{b}a\overline{b}$;	$\overline{a}]$
Gen. list 1:	$[a$;	$\overline{a}]$		
Gen. list 2:	$[b$;	$\overline{b}]$		
Gen. list 3:	$[\varepsilon$;	b;	$\overline{b}]$	
Gen. list 4:	$[\varepsilon$;	b;	$\overline{b}]$	
Gen. list 5:	$[\varepsilon$;	ε;	a;	$\overline{a}]$

Finally, the correct order to merge the generating list must be computed. Given the succession of the generating lists, start from the topmost generating list (gen. list 1), then attempt to compute a partial AntiDyck string that progressively reproduces the full AntiDyck string, by identifying and

extracting from the residual succession of the generating lists the topmost list that merges properly with the partial AntiDyck string computed so far, and so on until no generating list is left. The following table shows this permutation process.

Reordering of the generating lists					
Full AntiDyck list:					$[ab;\ \bar{a}bb\bar{b};\ \bar{b}a\bar{b};\ \bar{a}]$
	Components				Partial AntiDyck lists
Permuted gen. lists	1	2	3	4	Initial list: [ ]
Gen. list 1	$[a;$	$\bar{a}]$			$[a;\ \bar{a}]$
Gen. list 3	$[\varepsilon;$	$b;$	$\bar{b}]$		$[a;\ \bar{a}b;\ \bar{b}]$
Gen. list 5	$[\varepsilon;$	$\varepsilon;$	$a;$	$\bar{a}]$	$[a;\ \bar{a}b;\ \bar{b}a;\ \bar{a}]$
Gen. list 4	$[\varepsilon;$	$b;$	$\bar{b}]$		$[a;\ \bar{a}bb;\ \bar{b}a\bar{b};\ \bar{a}]$
Gen. list 2	$[b;$	$\bar{b}]$			$[ab;\ \bar{a}bb\bar{b};\ \bar{b}a\bar{b};\ \bar{a}]$

The whole process converges to the solution. This also shows that the AntiDyck language provides an unexpected link between very different formalisms.

6. HIERARCHY OF LIST LANGUAGES

The three families of list languages FLL, RLL and sRLL form a proper hierarchy. This section is entirely dedicated to prove such a fact. As a technical detail, we assume in the following that, given an alphabet Σ, the set $\Sigma_S = \Sigma \cup \{;\}$ is the same alphabet as Σ, but extended with the separator $(;)$, and that $\pi : \Sigma_S^* \to \Sigma^*$ is the natural projection cancelling the separator $(;)$.

STATEMENT 6.1 (Image): *Any RLL and any sRLL has a regular flattened image.*

Proof: Given a RLL $\underline{L} \subseteq \underline{\Sigma}^{[+]}$, its flattened image is $\backslash\underline{L}\backslash = \pi\,(string)\,(\underline{L})) \subseteq \Sigma^*$ (*see* statement 2.1). Since the language $string\,(\underline{L})$ is regular by the definition of RLL and π is a projection, also the flattened image of $\underline{L}$ is regular. If $\underline{L}$ is sRLL then the flattened image of $\underline{L}$ is precisely the regular language whose free segmentation is $\underline{L}$.

The next statement gives a decomposition property of RLL, useful for comparing RLL with FLL.

STATEMENT 6.2 (Decomposition): *Every non-void RLL* $\underline{R} \subseteq \underline{\Sigma}^{[+]}$ *has a word image string* $(\underline{R}) \subseteq \Sigma_S^*$ *that can be decomposed as follows:*

$$\left.\begin{array}{c} string\,(\underline{R}) = L\,(rexpr\,(\underbrace{R_1^S, \ldots, R_i^S, \ldots, R_n^S, (;)}_{\text{generators}})) \\ \text{where } R_i^S \subseteq \Sigma^* \quad \text{for } 1 \leq i \leq n \end{array}\right\} \tag{1}$$

where $rexpr$ *is a regular expression having as generators the non-void regular languages* R_i^S *and the separator* $(;)$.

Proof: The statement is proven by showing a construction for the regular expression $rexpr$ and the generators R_i^S.

Notational premise. Denote a deterministic FSA as follows: $A = (Q, \Sigma, q^-, F, \delta)$, where Q is the set of states, Σ is the input alphabet, $q^- \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta : \Sigma \times Q \to Q$ is the transition function.

Definition of relationship (1). Consider that since $\underline{R}$ is a non-void RLL there exists a non-void regular language $\underline{R}_S \subseteq \Sigma_S^*$ such that $R_S = string\,(\underline{R})$ (definition 3.3).

Take the deterministic FSA $A = (Q, \Sigma, q^-, F, \delta)$, acting over an input alphabet without separator and recognizing a regular language $R = L\,(A) \subseteq \Sigma^*$. A *free cut* automaton of A, of type (p, q) (where p, $q \in Q$), is the *maximal subautomaton* $A_{p,q} = (Q, \Sigma, p, \{q\}, \delta)$ of A, where the maximality is taken over the transitions; there exist at most $|Q|^2$ such free cut automata. Briefly, $A_{p,q}$ consists of all the paths in A from state p to state q. Denote as $R_{p,q} = L\,(A_{p,q}) \subseteq \Sigma^*$ the regular language recognized by the free cut automaton $A_{p,q}$, called free cut language of type (p, q), and denote as $C = \{R_{p,q} | p, q \in Q\} \subseteq \wp\,(\Sigma^*)$ the (finite) family of free cut languages.

Take the deterministic FSA $A_S = (Q, \Sigma_S, q^-, F, \delta)$, acting over an alphabet extended with the separator $(;)$ and recognizing the regular language $R_S = L\,(A_S) \subseteq \Sigma_S^*$. A *bounded cut* language $R_{p,q}^S \subseteq \Sigma^*$ is defined as the regular language $R_{p,q}^S = (R_{p,q} \cap (;\, \Sigma^*))/(;)$ (the slash "/" indicates the left quotient), where $R_{p,q} \in \Sigma_S^*$ is a free cut language of the automaton A_S (note that $R_{p,q}^S$ does not contain the separator $(;)$, while $R_{p,q}$ may contain it). Briefly, a bounded cut language $R_{p,q}^S$ consists of all the paths sharing the same initial and final state p and q, respectively, starting in p with a separator

$(;)$, which is cancelled, continuing after q with a separator $(;)$, but not containing any separator $(;)$ in the middle. Denote as $C_S = \{R^S_{p,q} | p, q \in Q$ and $R^S_{p,q} \neq \emptyset\} \subseteq \wp(\Sigma^*)$ the (finite) family of non-void bounded cut languages.

The following local constraints tie the bounded cut languages:

$$string(\underline{R}) = (C_S \cup \{;\})^* - P(; C_S)^* - (C_S ;)^* S - (C_S ;)^* M(; C_S)^*$$
$$P = \bigcup_{p \neq q^-} R^S_{p,q} \quad S = \bigcup_{q \notin F} R^S_{p,q} \quad M = \bigcup_{\delta((;),q) \neq r} R^S_{p,q}; R^S_{r,s} \qquad (2)$$

Define C_S as alphabet, whose elements are the "names" of the bounded cut languages $R^S_{p,q}$. After relationship (2), the regular language R_S is a 2-definite language, where the alphabet is C_S plus the separator $(;)$. Therefore R_S is generated by a regular expression $rexpr$, definable as follows:

$$string(\underline{R}) = L(rexpr(R^S_{p_1,q_1}, \ldots, R^S_{p_i,q_i}, \ldots, R^S_{p_n,q_n}, (;))) \qquad (3)$$

having as generators the elements of C_S and the separator $(;)$. We claim that relationship (3) is the relationship (1) whose existence is affirmed by the thesis of the statement.

Computation of the relationship (1): The computation of R^S_i is implicit in its definition, that of $rexpr$ can be found for instance in [Brs79], and consists in passing from a set of local constraints to the corresponding local regular language.

Proof of the correctness of relationship (1): Relationship (2) expresses the local constraints that guarantee the bounded cut languages R^S_i are concatenated together correctly to reconstruct the strings of R_S. In detail, the various terms of (2) have the following interpretations:

- $P(; C_S)^*$: the first bounded cut language starts at the initial state of A_S.
- $(C_S ;)^* S$: the last bounded cut languages end at some final state of A_S.
- $(C_S ;)^* M(; C_S)^*$: the ending and starting states of two consecutive bounded cut languages are connected by a transition reading the separator $(;)$ in the input of A_S.

Hence relationship (2) shows immediately that R_S is a regular 2-definite language over the alphabet $C_S \cup \{;\}$, containing the names R^S_i of the bounded cut languages and the separator $(;)$.

As the above list shows, the local constraints contained in relationship (2) are such that;

• The bounded cut languages R_i^S are concatenated in such a way that only strings of the language R_S are obtained.

• Any string of the language R_S is obtained by concatenating the bounded cut languages R_i^S in a way allowed by the regular expression $rexpr$.

Therefore relationship (3) has the properties claimed for relationship (1), and is formally equivalent to relationship (1) through a simple renaming of some indices. □

The following statement proves the existence of a proper hierarchy of regular list languages inside the family FLL.

STATEMENT 6.3 (Hierarchy): *It holds* $sRLL \subset RLL \subset FLL$, i.e. *simple regular, regular and fair list languages form a proper hierarchy*

Proof: The proof is organized by considering the two inclusions separately.

Inclusion $sRLL \subset RLL$: it follows directly from the definitions of sRLL and RLL. In fact, if $\mathcal{R}$ is the family of regular languages, then $\pi^{-1}(\mathcal{R}) \subseteq \mathcal{R}$, due to that regular languages are closed w.r.t. back-projection, hence $sRLL = list(\pi^{-1}(\mathcal{R})) \subseteq list(\mathcal{R}) = RLL$, because *list* is a one-to-one mapping (*see* statement 2.1). Moreover, there exists no regular language $R \subseteq \Sigma^*$ s.t. $\pi^{-1}(R) = \Sigma^*(;\Sigma^*;\Sigma^*)^*$, that is, the strings of the back-projection π^{-1} of R contain an even number of separators $(;)$, including none. In fact, the back-projection $\pi^{-1}(R)$ would unavoidably saturate the strings of R with respect to the introduction of the separator $(;)$.

Inclusion $RLL \subset FLL$: it is necessary to show that given any regular language $R_S \subseteq \Sigma_S^*$, the RLL $\underline{R} = list(R_S) \subseteq \underline{\Sigma}^{[+]}$ is generated by some FE.

Definition of the FE: it is defined by composing two different types of fair expressions, one not containing merge and the other not containing catenation; to this purpose, we use statement 6.2. The word image $string(\underline{R}) = R_S$ of the RLL $\underline{R}$ can be decomposed according to relationship (1):

$$string(\underline{R}) = L(rexpr(R_1^S, \ldots, R_i^S, \ldots, R_n^S, (;)))$$

where the regular languages $R_i^S \subseteq \Sigma^*$ are the bounded cut languages, for any $1 \leq i \leq n$. Denote as $cut_i^S(a_{i_1}, \ldots, a_{i_m})$ the regular expressions that generate the R_i^S's, with generators $a_{i_1}, \ldots, a_{i_m} \in \Sigma \cup \{\varepsilon\}$ (in the following we shall omit the generator, for brevity), respectively, *i.e.* $R_i^S = L(cut_i^S)$.

By statement 6.2 the projected regular language $\pi(R_S) \subseteq \Sigma^*$, whose strings are those of R_S but with all separators $(;)$ cancelled, is generated

by the following regular expression:

$$\begin{aligned}\pi(R_S) &= \pi(L(rexpr(R_1^S, \ldots, R_i^S, \ldots, R_n^S, (;)))) \\ &= L(\pi(rexpr(R_1^S, \ldots, R_i^S, \ldots, R_n^S)) \\ &= L(paste(R_1^S, \ldots, R_i^S, \ldots, R_n^S,)) \end{aligned} \tag{4}$$

where $paste = \pi(rexpr)$ is a regular expression.

To obtain the FE that generates $\underline{R}$, we transform the compound regular expression (4) into a FE, by using the two mappings μ and ν, as follows:

$$\mu : reg.expr. \to fair\ expr. \qquad \nu : reg.expr. \to fair\ expr.$$

$$\begin{aligned} generator &\overset{\mu}{\mapsto} [generator] & generator &\overset{\nu}{\mapsto} [generator] \\ s_1 \cdot s_2 &\overset{\mu}{\mapsto} \mu(s_1) \| \mu(s_2) & r_1 \cdot r_2 &\overset{\nu}{\mapsto} \nu(r_1) \bullet \nu(r_2) \\ s^+ &\overset{\mu}{\mapsto} \mu(s)^{\|+} & r^+ &\overset{\nu}{\mapsto} \nu(r)^+ \\ s^* &\overset{\mu}{\mapsto} [\varepsilon] \cup \mu(s)^{\|*} & r^* &\overset{\nu}{\mapsto} [\varepsilon] \cup \nu(r)^* \end{aligned}$$

where s, s_1 and s_2 are regular expressions over the alphabet Σ, *i.e.* $generator \in \Sigma \cup \{\varepsilon\}$, and r, r_1 and r_2 are regular expressions over the alphabet C_S, *i.e.* $generator \in C_S$.

We denote $\underline{paste} = \nu(paste)$ and $\underline{cut_i^S} = \mu(cut_i^S)$. The image FE's $\underline{cut_i^S}$ have as generators one-component one-letter lists of the type $[a]$ and $[\varepsilon]$ with $a \in \Sigma$; the image FE $\underline{paste}$ has as generators one-component one-letter lists of the type $[\overline{R_i^S}]$ with $\overline{R_i^S} \in C_S$. The image FE's $\underline{cut_i^S}$ generate one-component lists, *i.e.* $[w]$ with $w \in \Sigma^*$; the image FE $\underline{paste}$ generates lists with components consisting of one letter, *i.e.* $[\overline{R_{i_1}^S}; \ldots; \overline{R_{i_n}^S}]$ with $R_{i_i}^S \in C_S$. We claim that the FE:

$$f = \underline{paste}(\underline{cut_1^S}, \ldots, \underline{cut_i^S}, \ldots, \underline{cut_n^S}) \tag{5}$$

generates $\underline{R}$, *i.e.* $\underline{R} = L(f)$.

Computation of the FE (5): it is necessary to compute the FE's $\underline{cut_i^S}$ and $\underline{paste}$.

The regular expressions cut_i^S are computed by first cutting away the subautomaton A_i^S from the whole automaton A_S, by intersecting it with $(; \Sigma^*)$, by computing its equivalent regular expression and finally by

quotienting away on the left the generator $(;)$. The FE's $\underline{cut}_i^S$ are computed by applying the mapping μ.

The regular expression *paste* is computed by projecting through π the regular expression $rexpr$ of relationship (1); obviously *paste* does not contain the generator $(;)$ any longer. The FE $\underline{paste}$ is computed by applying the mapping ν.

Proof of the correctness of the FE (5): we must prove that the FE f generates $\underline{R}$. To this purpose, we analyze the behaviour of the subexpressions $\underline{cut}_i^S$ and $\underline{paste}$; statement 3.1 is used.

$$\backslash \underline{cut}_i^S \backslash = \backslash \mu \left(cut_i^S \right) \backslash = L \left(cut_i^S \right) = R_i^S \tag{6}$$

The derivation (6) holds because: flattening is an isomorphism between FE restricted to union, merge, merge closure and one-component generators, and regular expressions (*see* statement 3.1 for the proof); and flattening and the mapping μ are each other's inverse, since flattening is here one-to-one, *see* statement 3.1, whence μ is here one-to-one, and flattening and μ revert each other's operations. Symbolically, we can rewrite relationship (6) as $L \left(\underline{cut}_i^S \right) = [R_i^S]$.

$$\begin{aligned}
\backslash \underline{paste} \left(\underline{cut}^S \right) \backslash &= \backslash \underline{paste} \left(\underline{cut}_1^S, \ldots, \underline{cut}_i^S, \ldots, \underline{cut}_n^S \right) \backslash \\
&= \backslash \underline{paste} \left[R_1^S \right], \ldots, \left[R_i^S, \ldots, \left[R_n^S \right] \right) \backslash \\
&= \backslash \nu \left(paste \left(R_1^S, \ldots, R_i^S, \ldots, R_n^S \right) \right) \backslash \\
&= L \left(paste \left(\backslash [R_1^S] \backslash, \ldots, \backslash [R_i^S] \backslash, \ldots, \backslash [R_n^S] \backslash \right) \right) \\
&= L \left(paste \left(\backslash \underline{cut}_1^S \backslash, \ldots, \backslash \underline{cut}_i^S \backslash, \ldots, \backslash \underline{cut}_n^S \backslash \right) \right)
\end{aligned} \tag{7}$$

The derivation (7) holds because: flattening is an isomorphism between FE restricted to union, catenation, catenation closure and generators with one-letter components, and regular expressions (*see* statement 3.1 for the proof); and flattening and the mapping ν are each other's inverse, since flattening is here one-to-one, *see* statement 3.1, whence ν is here one-to-one, and flattening and ν revert each other's operations.

$$\begin{aligned}
\backslash f \backslash &= \backslash \underline{paste} \left(\underline{cut}_1^S, \ldots, \underline{cut}_i^S, \ldots, \underline{cut}_n^S \right) \backslash \\
&= L \left(paste \left(\backslash \underline{cut}_1^S \backslash, \ldots, \backslash \underline{cut}_i^S \backslash, \ldots, \backslash \underline{cut}_n^S \backslash \right) \right) \\
&= L \left(paste \left(R_1^S, \ldots, R_i^S, \ldots, R_n^S \right) \right) \\
&= \pi \left(R_S \right) = \backslash \underline{R} \backslash
\end{aligned} \tag{8}$$

The final derivation (8) holds because of: relationship (7); relationship (6); and relationship (4). So we have that $\backslash\underline{R}\backslash = \backslash f\backslash$; but since flattening is one-to-one on expressions of the types $\underline{cut}$ and $\underline{paste}$, whence also on expressions of type f, it immediately follows that $\underline{R} = L(f)$.

The strictness of inclusion $RLL \subset FLL$ follows by observing that, for instance, the language $\underline{AntiDyck(\Sigma)}$ of example 5.1 is a FLL but not a RLL, as its flattened image is not a regular language. □

Statement 6.3 admits a weaker version, that works for sRLL instead of RLL. If the regular language R is recognized by an automaton A, acting over an alphabet Σ without separator, then it suffices to replace systematically the bounded cut languages with the free cut languages and to omit the separator $(;)$ in the relationship (2).

We want to show an example of the construction involved in statement 6.3. As it is fairly complex, mainly due to the computation of the regular expressions generating the bounded cut languages, we turn to show the simpler case for sRLL, which still embeds all of it essence. With the help of intuition, the involved regular expressions can be inferred through a quick look at the state diagram of the automaton A.

Example 6.1 (sRLL): Take the deterministic FSA $A = (\{a, b\}, \{q_1, q_2\}, q_1, \{q_2\}; \delta)$ represented in Figure 2. Clearly one has $R = L(A) = a^* b \Sigma^*$, where $\Sigma = \{a, b\}$ is the input alphabet of A. There exist four free cut automata, represented in Figure 3 (they are neither reduced nor minimized). They are obtained from A by simply reassigning the initial state (dangling arrow) and final states (shading); it is immediate to compute their equivalent regular expressions. The cut regular expressions and their FE's are easily seen to be, posing $\underline{\Sigma} = \{[a], [b]\}$:

$$
\begin{array}{lcl}
R_{1,1} = L(A_{1,1}) = a^* & \overset{\mu}{\mapsto} & \underline{cut}_{1,1} = [\varepsilon] \cup [a]^{\|*} \\
R_{1,2} = L(A_{1,2}) = a^* b \Sigma^* = R & \overset{\mu}{\mapsto} & \underline{cut}_{1,2} = ([\varepsilon] \cup [a]^{\|*}) \| [b] \| ([\varepsilon] \cup \underline{\Sigma}^{\|*}) \\
R_{2,1} = L(A_{2,1}) = \emptyset & \overset{\mu}{\mapsto} & \underline{cut}_{2,1} = \emptyset \\
R_{2,2} = L(A_{2,2}) = \Sigma^* & \overset{\mu}{\mapsto} & \underline{cut}_{2,2} = [\varepsilon] \cup \underline{\Sigma}^{\|*}
\end{array}
$$

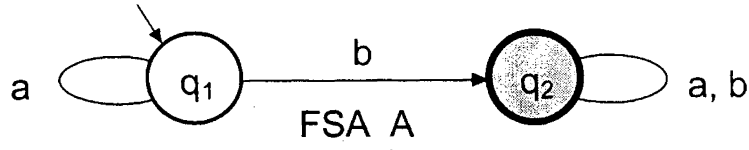

Figure 2. – Deterministic FSA to be freely segmented.

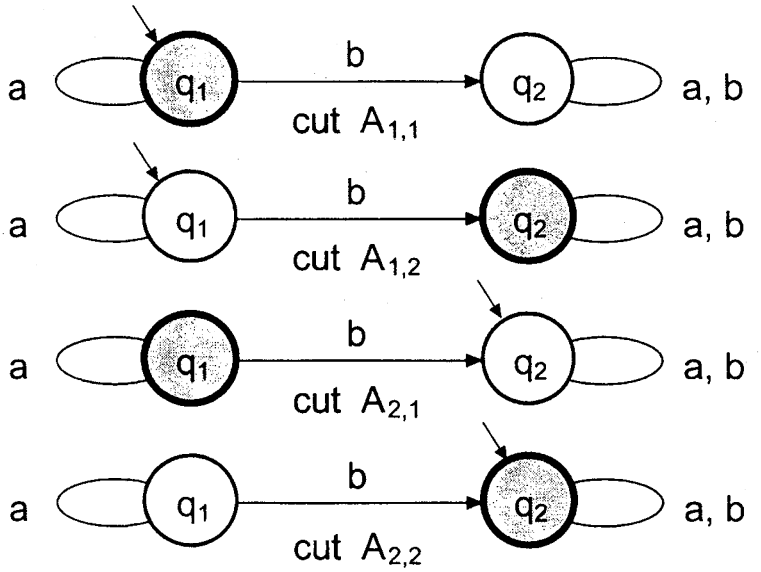

Figure 3. – Free cut automata obtained from the deterministic FSA A

The regular expression *paste* and the FE $\underline{paste}$ are (we minimize relationship (2) for *paste*, observing that $R_{2,1} = \emptyset$):

$$R = R_{1,1}^{*}\, R_{1,2}\, R_{2,2}^{*} \overset{\nu}{\mapsto} \underline{paste} = (\underline{cut}_{1,1})^{*}\, \underline{cut}_{1,2}\, (\underline{cut}_{2,2})^{*}$$

Finally, the FE f that generates the sRLL $\backslash R \backslash^{-1}$ is:

$$f = ([\varepsilon] \cup [a]^{\|*})_1^{*}\, (([\varepsilon] \cup [a]^{\|*})_3 \| [b] \| ([\varepsilon] \cup \underline{\Sigma}^{\|*})_4)_2\, ([\varepsilon] \cup \underline{\Sigma}^{\|*})_5^{*} \quad \underline{\Sigma} = \{[a], [b]\}$$

For instance, the expansion for lists of type $[a^{*};\, a^{m}\, ba^{n};\, \varepsilon;\, \ldots;\, \varepsilon]$ $(m, n \geq 0)$ is:

$$\begin{aligned}&[a^{*};\, a^{m}\, ba^{n};\, \varepsilon;\, \ldots;\, \varepsilon] \\ &\quad = ([a] \| \ldots \| [a])_1\, ((\underbrace{[a] \| \ldots \| [a]}_{m \text{ times}})_3 \| [b] \| (\underbrace{[a] \| \ldots \| [a]}_{n \text{ times}})_4)_2\, ([\varepsilon])_5^{*}\end{aligned}$$

The two FE's f_9 and f_{10} of example 4.6 are derived by applying the above construction, with some final simplification.

7. PROPERTIES OF LIST LANGUAGES

In this section, the language-theoretic properties of the various families of list languages are investigated, including semilinearity, closure and decidability. As a technical detail, we assume that Σ is the alphabet, Σ_S is the same alphabet as Σ, but extended with a separator $(;)$, and $\pi : \Sigma_S^* \to \Sigma^*$ is the natural projection cancelling the separator $(;)$.

7.1. Semilinearity

The family fFLL inherits the semilinearity property [Gin75] from regular languages.

STATEMENT 7.1 (Semilinearity): *Let $\Psi : \Sigma^* \to N^{|\Sigma|}$ be the Parikh function of strings* [Har78, Sal73]. *Then the image space of the family fFLL through Ψ is semilinear.*

Proof: Observe that FLL is letter-equivalent to a regular language, dropping the separators $(;)$. In fact, it suffices to take a FE, drop the separators $(;)$ and map both merge and list catenation onto string catenation. This produces a regular expressions generating strings that are letter-equivalent to the lists generated by the FE, although the letters are more or less permuted. Letter-equivalent languages have the same Parikh image and regular languages are semilinear. □

The Parikh image is definable also for FLL, and coincides with that of fFLL, but not including the separator $(;)$.

7.2. Closure properties

Now we can prove the main closure properties of FLL, fFLL, RLL and sRLL. As usual Σ_S denotes the alphabet Σ extended with the separator $(;)$.

7.2.1 *Homomorphism*

Direct and reverse homomorphisms for lists are defined similarly to strings.

DEFINITION 7.1 (List Homomorphism): *The function $\underline{\varphi} = \underline{\Sigma}^{[*]} \to \underline{\Delta}^{[*]}$ is a list homomorphism on lists, over the alphabet Σ, to lists, over the alphabet Δ, if and only if, for any lists $\underline{x}$, $\underline{y} \in \underline{\Sigma}^{[*]}$, the following holds:*

$$\underline{\varphi}(\underline{x}\|\underline{y}) = \underline{\varphi}(\underline{x})\|\underline{\varphi}(\underline{y}) \qquad \underline{\varphi}(\underline{x}\underline{y}) = \underline{\varphi}(\underline{x})\,\underline{\varphi}(\underline{y}) \qquad \underline{\varphi}([\ ]) = [\ ]$$

Reverse list homomorphism $\underline{\varphi}^{-1}$ is defined in the obvious way, *i.e.* $\underline{\varphi}^{-1}(\underline{x}) = \{\underline{y} | \underline{\varphi}(\underline{y}) = \underline{x}\}$. It is easy to see that if $\varphi : \Sigma^* \to \Delta^*$ is a

word homomorphism and the function $\underline{\varphi} : \underline{\Sigma}^{[*]} \rightarrow \underline{\Delta}^{[*]}$ is defined through the following mapping:

$$\underline{\varphi}([x_1; x_2; \ldots; x_n]) = [\varphi(x_1); \varphi(x_2); \ldots; \varphi(x_n)] \quad x_i \in \Sigma^* \quad \text{for } 1 \leq i \leq n \tag{9}$$

and moreover posing $\underline{\varphi}([\ ]) = [\ ]$, then $\underline{\varphi}$ is a list homomorphism. It is said that the list homomorphism $\underline{\varphi}$ is *associated* with the word homomorphism φ. A similar association holds for reverse homomorphism, *i.e.* $\underline{\varphi}^{-1}([x_1; x_2; \ldots; x_n]) = \{[y_1; y_2; \ldots; y_n] | y_1 \in \varphi^{-1}(x_1), y_2 \in \varphi^{-1}(x_2), \ldots, y_n \in \varphi^{-1}(x_n)\}$. List homomorphisms always come in association with word homomorphisms.

STATEMENT 7.2 (Association): *Direct and reverse list homomorphisms are one-to-one associated with direct and reverse word homomorphisms, respectively.*

Proof: A word homomorphism always induces a list homomorphism, by relationship (9). Conversely, let $\underline{\varphi}$ be a generic list homomorphism. Take any two one-component lists $[x]$, $[y]$, where $x, y \in \Sigma^*$ are two generic strings, then it holds: $\underline{\varphi}([x]||[y]) = \underline{\varphi}([x])||\underline{\varphi}([y])$ and $\underline{\varphi}([x]||[y]) = \underline{\varphi}([xy])$. Therefore $\underline{\varphi}$ works on one-component lists like a string homomorphism. Take now any multiple-component list $[x_1; x_2; \ldots; x_n]$, where the $x_i \in \Sigma^*$ are generic strings for $1 \leq i \leq n$, then it holds $\underline{\varphi}([x_1; \ldots; x_n]) = \underline{\varphi}([x_1][x_2] \ldots [x_n]) = \underline{\varphi}([x_1])\underline{\varphi}([x_2]) \ldots \underline{\varphi}([x_n])$. Since $\underline{\varphi}$ works like a string homomorphism on one-component lists, is must work on multiple-components lists as specified by relationship (9). The same reasoning applies to reverse homomorphism. Association is clearly a one-to-one correspondence. □

A consequence of statement 7.2 is that, given any list $\underline{x} \in \underline{\Sigma}^{[*]}$, the number of components of $\underline{\varphi}(x)$ is the same as that of $\underline{x}$; hence if $\underline{\varphi}(x) = [\ ]$ it follows that $\underline{x} = [\ ]$.

A generic list homomorphism is said to be *arbitrary*. A list homomorphism is said to be *alphabetic* if and only if it maps characters onto characters, *i.e.* iff $\underline{\varphi}([a]) = [b]$, where $a, b \in \Sigma$. A list homomorphism is said to be *erasing* if and only if it can map some character to ε, *i.e.* iff $\underline{\varphi}([a]) = [\varepsilon]$, for some $a \in \Sigma$. It is easy to see that list homomorphisms are alphabetic (resp. erasing) if and only if they are associated with alphabetic (resp. erasing) word homomorphisms. A reverse list homomorphism is said to be arbitrary (resp. alphabetic, erasing) if and only if the corresponding direct list homomorphism is arbitrary (resp. alphabetic, erasing).

STATEMENT 7.3 (Commutation): *Arbitrary direct and alphabetic reverse homomorphisms commute with flattening,* i.e.*:*

$$\backslash\underline{\varphi}(x)\backslash = \varphi(\backslash\underline{x}\backslash) \qquad \textit{and} \qquad \backslash\underline{\psi}^{-1}(x)\backslash = \psi^{-1}(\backslash\underline{x}\backslash)$$

where $\underline{x}$ *is a list,* $\underline{\varphi}$ *and* $\underline{\psi}^{-1}$ *are arbitrary direct and alphabetic reverse list homomorphisms, respectively, and* φ, ψ^{-1} *are their associated arbitrary direct and alphabetic reverse word homomorphisms, respectively.*

Proof: Statement 7.2 implies that:

$$\begin{aligned}
\backslash\underline{\varphi}(x)\backslash &= \backslash[\varphi(x_1);\ \varphi(x_2);\ \ldots;\ \varphi(x_n)]\backslash = \\
&= \varphi(x_1)\,\varphi(x_2)\,\ldots\,\varphi(x_n) \\
&= \varphi(x_1\,x_2\,\ldots\,x_n) = \varphi(\backslash\underline{x}\backslash) \\
\backslash\underline{\psi}^{-1}(\underline{x})\backslash &= \backslash[\psi^{-1}(x_1);\ \psi^{-1}(x_2);\ \ldots,\ \psi^{-1}(x_n)]\backslash \\
&= \psi^{-1}(x_1)\,\psi^{-1}(x_2)\,\ldots\,\psi^{-1}(x_n) \\
&= \psi^{-1}(x_1\,x_2\,\ldots\,x_n) = \psi^{-1}(\backslash x\backslash)
\end{aligned}$$

For the passage $\psi^{-1}(x_1)\,\psi^{-1}(x_2)\,\ldots\,\psi^{-1}(x_n) = \psi^{-1}(x_1\,x_2\,\ldots\,x_n)$ it is essential that reverse homomorphism is alphabetic. □

By a standard proof [Brs79], arbitrary direct and alphabetic reverse list homomorphisms commute also with catenation and merge, and hence also with their closures. Now we can prove the homomorphism closure properties in detail.

STATEMENT 7.4 (Homomorphism): *The families FLL and fFLL are closed w.r.t. arbitrary direct homomorphism and alphabetic reverse homomorphism. The families RLL (resp. sRLL) are closed w.r.t. arbitrary (resp. alphabetic) direct and arbitrary (resp. alphabetic) reserve homomorphism.*

Proof: For a generic FLL $\underline{L}$ observe that the closure w.r.t. any arbitrary direct list homomorphism or alphabetic reverse list homomorphism $\underline{\varphi}$ or $\underline{\psi}^{-1}$, respectively, holds because it suffices to apply $\underline{\varphi}$ or $\underline{\psi}^{-1}$, respectively, to the generators of the FE that generates the language $\underline{L}$; for the construction to work correctly it is essential that the reverse list homomorphism $\underline{\psi}^{-1}$ is alphabetic [Brs79].

For fFLL statement 7.3 ensures that arbitrary direct and alphabetic reverse homomorphisms commute with flattening, so the closure follows from that of FLL.

The proof for RLL and sRLL is the following. Let $\varphi : \Sigma_S^* \to \Delta_S^*$ be an arbitrary word homomorphism that preserves the separator $(;)$, *i.e.*

$\varphi(;) = (;)$, and let $\underline{\varphi} : \underline{\Sigma}^{[*]} \to \underline{\Delta}^{[*]}$ be its associated arbitrary list homomorphism. Let now $R \subseteq \Sigma_S^*$ be a regular language over the alphabet Σ extended with the separator $(;)$. Then one has:

$$\underline{\varphi}(list(R)) = list(\varphi(R)) \qquad \text{and} \qquad \underline{\varphi}^{-1}(list(R)) = list(\varphi^{-1}(R))$$

because *list* is one-to-one and $\underline{\varphi}$ is associated with φ. Let now $R \subseteq \Sigma^*$ be a regular language and let φ be alphabetic. Then one has:

$$\underline{\varphi}(\backslash R \backslash^{-1}) = \backslash \varphi(R) \backslash^{-1} \qquad \text{and} \qquad \underline{\varphi}^{-1}(\backslash R \backslash^{-1}) = \backslash \varphi^{-1}(R) \backslash^{-1}$$

because free segmentation $\backslash\backslash^{-1}$ preserves characters and $\underline{\varphi}$ is alphabetic and associated with φ. These relationships prove the homomorphism closures of RLL and sRLL. □

The family sRLL is clearly not closed w.r.t. direct and reverse arbitrary homomorphism.

7.2.2. *Boolean closures*

Boolean closures are defined as usual; only note that, since RLL and sRLL are [ ]-free, complement must be taken with respect to the [ ]-free universal list language. The next statement extends to list languages the important Franchi Zannettacci-Vauquelin theorem [Fra80] on the AntiDyck language.

STATEMENT 7.5 (Generalized Franchi Zannettacci-Vauquelin Th.): *A word language is recursively enumerable if and only if it is the flattened image of a homomorphic image of the intersection of the list AntiDyck language* $\underline{AntiDyck}$ (see example 5.1) *with a sRLL,* i.e.*:*

$$E = \backslash \underline{\varphi}\, \underline{AntiDyck} \cap \underline{R}) \backslash$$

where $E \subseteq \Sigma^*$ *is a recursively enumerable language,* $\underline{\varphi} : \underline{\Delta}^{[*]} \to \underline{\Sigma}^{[*]}$ *is an arbitrary list homomorphism,* $\underline{AntiDyck}(\Sigma) \subseteq \underline{\Delta}^{[*]}$ *is the list AntiDyck language and* $\underline{R} \subseteq \underline{\Delta}^{[+]}$ *is a sRLL.*

Proof: The proof is an extension to list languages of the well-known Franchi Zannettacci-Vauquelin theorem on the AntiDyck language [Fra80]: "*a language is recursively enumerable if and only if it is a homomorphic image of the intersection of the AntiDyck language* (see definitions 5.1 and 5.2) *with a regular language*". In symbols, given an alphabet Σ and posing $\Delta = \Sigma \cup \overline{\Sigma}$ where $\overline{\Sigma}$ is a disjoint copy of Σ, let $E \subseteq \Sigma^*$ represent a recursively enumerable language, $\varphi : \Delta^* \to \Sigma^*$ an arbitrary

homomorphism, $AntiDyck(\Sigma) \subseteq \Delta^*$ the AntiDyck language and $R \subseteq \Delta^*$ a regular language, then the following is an identity:

$$E = \varphi(AntiDyck(\Sigma) \cap R) \qquad \text{(Franchi Zannettacci-Vauquelin [Fra80])} \tag{10}$$

We can now extend relationship (10) to list languages, searching a list language $\underline{E}$ whose flattened image will be E. Construct the new list language $\underline{E}$ as follows: $\underline{E} = \underline{\varphi}(\underline{AntiDyck}(\Sigma) \cap \underline{R}) \subseteq \underline{\Sigma}^{[*]}$. Of the objects appearing in the formula, the list language $\underline{R} \subseteq \underline{\Delta}^{[+]}$ is a sRLL defined as $\underline{R} = \backslash R \backslash^{-1}$ and the arbitrary list homomorphism $\underline{\varphi} : \underline{\Delta}^{[*]} \to \underline{\Sigma}^{[*]}$ is that associated with the arbitrary word homomorphism φ. The list language $\underline{E}$ is the searched one, since the equality $E = \backslash \underline{E} \backslash$ holds, because:

$$\begin{aligned}
\backslash \underline{E} \backslash &= \backslash \underline{\varphi}(\underline{AntiDyck}(\Sigma) \cap \underline{R}) \backslash && (11)\\
&= \varphi(\backslash \underline{AntiDyck}(\Sigma) \cap \underline{R} \backslash) && (12)\\
&= \varphi(\pi(string(\underline{AntiDyck}(\Sigma) \cap \underline{R}))) && (13)\\
&= \varphi(\pi(string(\underline{AntiDyck}(\Sigma)) \cap string(\underline{R}))) && (14)\\
&= \varphi(\pi(string(\underline{AntiDyck}(\Sigma)) \cap \pi^{-1}(R))) && (15)\\
&= \varphi(\pi(string(\underline{AntiDyck}(\Sigma))) \cap R) && (16)\\
&= \varphi(\backslash \underline{AntiDyck}(\Sigma) \backslash \cap R) && (17)\\
&= \varphi(\underline{AntiDyck}(\Sigma) \cap R) = E && (18)
\end{aligned}$$

The passages in the equality are orderly based on: (11) the commutativity between flattening and arbitrary direct homomorphism (statement 7.3), (12) the functional identity $\backslash\backslash = \pi \circ string$ (statement 2.1), (13) the commutativity between the compaction function *string* and intersection holding since *string* is one-to-one, (14) the language identity $string(\underline{R}) = \pi^{-1}(R)$ holding since $\underline{R}$ is a sRLL (definition 3.4), (15) the functional identity $\alpha(U \cap \alpha^{-1}(V)) = \alpha(U) \cap V$ where α is a function and U, V are two sets proved in [Brs79] (in the present case the role of α is played by the projection π), (16) again the same functional identity $\pi \circ string = \backslash\backslash$ as in (12), (17) the language equality $AntiDyck(\Sigma) = \backslash \underline{AntiDyck}(\Sigma) \backslash$ (example 5.1), and eventually (18) relationship (10) which is the Franchi Zannettacci-Vauquelin theorem of [Fra80]. □

The Franchi Zannettacci-Vauquelin theorem and hence statement 7.5 hold even if the homomorphism is alphabetic and the regular language is local 2-definite [Fra80].

STATEMENT 7.6 (Intersection with Reg. List Lang.): *The family FLL is not closed w.r.t. the intersection with the family sRLL and the family fFLL is not closed w.r.t. the intersection with regular languages.*

Proof: We prove the statement by refutation. Assume that FLL is closed w.r.t. the intersection with sRLL. But by statement 7.5 every recursively enumerable language is the flattened image of the homomorphic image of the intersection of the list AntiDyck language with a sRLL. We know by statement 7.4 that FLL is closed w.r.t. arbitrary homomorphism and by example 5.1 that the list AntiDyck language is FLL. Were FLL closed w.r.t. the intersection with sRLL, by statement 7.5 it would follow that every recursively enumerable word language should be FLL. By statement 7.1 every FLL is semilinear, hence also such a word language should be semilinear. This is the contradiction disproving the initial assumption, since recursively enumerable languages are not semilinear, in general. The family fFLL is not closed w.r.t. the intersection with regular languages since fFLL and regular languages are the flattened images of FLL and sRLL, respectively. □

The above proof is compact, but it does not produce an explicit example. The following construction shows a simple one, instancing the proof of statement 7.5.

Example 7.1 ($COPY_\infty$): We shall show how to construct the language:

$$E = \{u_1\, u_2\, \ldots\, u_n | u_1 \in \Sigma^+\, u_i = u_{i-1} \qquad \text{for} \quad 2 \leq i \leq n\} \subseteq \Sigma^+$$

The language E is also called the $COPY_\infty$ language in the literature [Bra88], as it is the free replication of any string in Σ^+; it is not semilinear because the number of replications $n \geq 2$ is unbounded, and hence it cannot be fFLL. The language E is interesting as, in some sense, it lies at the border between semilinearity and non-semilinearity; in fact, were the number n of iterations of the prefix u_1 upper bounded, then the language E would be semilinear. We shall instance the construction of statement 7.5 by giving the regular language R and the homomorphism φ.

Take a (natural) alphabet Σ and let $\Sigma_S = \Sigma \cup \{\bot\}$ be the (natural) alphabet Σ extended with the separator $\bot$. Let $\overline{\Sigma}$ and $\overline{\Sigma}_S$ be their barred copies (so $\bot \in \Sigma_S$ and $\overline{\bot} \in \overline{\Sigma}_S$). Let also be $\Delta = \Sigma \cup \overline{\Sigma}$ and $\Delta_S = \Sigma_S \cup \overline{\Sigma}_S$, and let $\theta : \Delta_S^* \to \Sigma^*$ be the natural projection; θ cancels all the barred characters (including $\overline{\bot}$ and also the separator $\bot$). Take the AntiDyck language over the alphabet Δ_S, *i.e.* $AntiDyck(\Sigma_S) \subseteq \Delta_S^*$. Define the regular language:

$$R = \Sigma^+ (\bot T^+ \overline{\bot})^+ \bot \overline{\Sigma}^+ \overline{\bot} \subseteq \Delta_S^+$$

where T is a finite set of two-letter strings:

$$T = \{\overline{c}c | c \in \Sigma\} \subseteq \Delta^2$$

We claim that $E = \theta\,(AntiDyck\,(\Sigma_S) \cap R) = \{u_1\,u_2\,\ldots\,u_n | u_1 \in \Sigma^+\, u_i = u_{i-1} \text{ for } 2 \leq i \leq n\} \subseteq \Sigma^+$. $\square$

In fact, the strings of the regular language R are formed by segments, over the alphabet Δ, separated by two-letter factors of the type $\perp\,\overline{\perp}$ (but the leftmost separator $\perp$ and its rightmost barred copy $\overline{\perp}$, that may occur isolated). Recall the interpretation of the AntiDyck language in terms of parentheses (open in Σ_S and closed in $\overline{\Sigma}_S$). Then one has that the intersection with R imposes to AntiDyck the following additional structure:

1. AntiDyck strings $u \in \Delta_S^*$ are segmented as follows:

$$u = u_1 \perp u_2 \overline{\perp}\perp u_3 \overline{\perp}\perp \ldots \overline{\perp}\perp u_{n-1} \overline{\perp}\perp u_n \overline{\perp}\perp u_{n+1} \overline{\perp}$$

for some integer $n \geq 2$.

2. The segment $u_1 \in \Sigma^+$, hence it is a string made only of open parentheses.

3. The segment $u_{n+1} \in \overline{\Sigma}^+$, hence it is a string made only of closed parentheses.

4. The segments $u_2, \ldots, u_n \in T^+$ with $n \geq 2$, hence they are strings made both of open and closed parentheses, but must exhibit a structure of the type:

$$u_i = \overline{c}_{i_1}\, c_{i_1}\, \overline{c}_{i_2}\, c_{i_2}\, \ldots\, \overline{c}_{i_n}\, c_{i_n} \qquad c_{i_i} \in \Sigma \quad 2 \leq i \leq n$$

where the natural character c is the copy of the barred character $\overline{c}$. So the segments u_i are formed by barred characters alternated to natural ones, and each barred character is always followed by its natural copy.

Constraints (1), (2), (3) and (4) impose to the internal segments of the AntiDyck strings a structure of the type $\overline{c}c\overline{d}d \ldots$, *i.e.* the order of the open parentheses must replicate that of the closed ones they follow. Conversely, AntiDyck itself implies that the order of the closed parentheses must replicate that of the open ones they are paired to. Intersection conjuncts both sets of constraints. The structure of the initial segment u_1 is totally free; any initial ordering of open parentheses is allowed. The structure of the following segments is determined by the initial one. The final result is such that, by delecting all the closed parentheses and the separator $\perp$ (through the projection θ), what remains are strings consisting of the replication of the initial segment u_1 for $n \geq 2$ times (note the similarity with the segmentation procedure explained in section 5).

For instance, $ab \perp \overline{a}ba\overline{b}\,\overline{\perp}\perp \overline{b}\overline{a}\overline{\perp}$ is filtered away by R, as it contains the substring $\overline{a}b$ (of type $\overline{c}d$ with $c \neq d$), $ab \perp \overline{a}aa\overline{b}\,\overline{\perp}\perp \overline{a}\overline{a}\,\overline{\perp}$ is filtered away by R, as it contains the substring $\overline{a}aa$ (of type $\overline{c}dv$ with $v \in \Sigma^+$), whereupon $ab \perp \overline{a}a\overline{b}b\,\overline{\perp}\perp \overline{a}\,\overline{b}\,\overline{\perp}$ is accepted by R; note that deleting all the barred characters and the separator $\perp$ yields $abab$, *i.e.* the replication of ab.

STATEMENT 7.7 (Boolean): *The families FLL and fFLL are closed w.r.t. union, but are not closed w.r.t. intersection and complement. The families RLL and sRLL are Boolean algebrae.*

Proof: The closure of FLL and of fFLL w.r.t. union is a consequence of the definition of FE and of that flattening commutes with union.

The families FLL and fFLL are not close w.r.t. intersection with sRLL and regular languages, respectively, by statement 7.6. Obviously such a closure fails also w.r.t. RLL, since $sRLL \subset RLL$ by statement 6.3. As $sRLL \subset FLL$ and hence regular languages are contained in fFLL, FLL and fFLL are not intersection closed. But FLL and fFLL are union closed. Were FLL and fFLL complement closed, De Morgan law would imply a contradiction with intersection non-closure, hence FLL and fFLL are not complement closed.

As for the Boolean closures for RLL, let R, R_1, $R_2 \subseteq \Sigma_S^*$ be any regular languages over an alphabet Σ, extended with the separator $(;)$; then it holds:

$$list\,(R_1) \cup list\,(R_2) = list\,(R_1 \cup R_2) \qquad \text{and} \qquad \overline{list\,(R)} = list\,(\overline{R})$$

Union and complement commute with the one-to-one mapping *list*. Intersection follows from union, complement and De Morgan laws.

As for the Boolean closures of sRLL, let R, R_1, $R_2 \subseteq \Sigma^*$ be any regular languages:

$$\backslash R_1 \backslash^{-1} \cup \backslash R_2 \backslash^{-1} = \backslash R_1 \cup R_2 \backslash^{-1} \qquad \text{and} \qquad \overline{\backslash R \backslash^{-1}} = \backslash \overline{R} \backslash^{-1}$$

Union and complement commute with the inverse function $\backslash\backslash^{-1}$. Intersection follows from union, complement and De Morgan laws. □

7.2.3. *Other closures*

We consider here some other closure properties of the families of list languages.

STATEMENT 7.8 (Catenation): *The families FLL and fFLL are closed w.r.t. catenation and catenation closure. The family RLL is closed w.r.t. catenation*

and [ ]*-free catenation closure. The family sRLL is not closed w.r.t. catenation and* [ ]*-free catenation closure.*

Proof: For FLL the statement follows from the definition of FE. For fFLL the statement follows from the catenation closure of FLL and by observing that flattening commutes with catenation and catenation closure, *i.e.* $\backslash \underline{L}_1 \bullet \underline{L}_2 \backslash = \backslash \underline{L}_1 \backslash \cdot \backslash \underline{L}_2 \backslash$ and $\backslash \underline{L}^* \backslash = \backslash \underline{L} \backslash^*$. For RLL, let R, R_1 and R_2 be any regular languages over the alphabet Σ_S containing a separator. Then one easily has:

$$list(R_1) \bullet list(R_2) = list(R_1; R_2) \qquad \text{and} \qquad list(R)^+ = list(R(; R)^*)$$

These relationships prove the statement for RLL, as regular languages are closed under catenation and Kleene star.

As for sRLL, let $R_1 = \{a\}$ and $R_2 = \{b\}$ be finite languages, hence regular, then one has:

$$\begin{aligned} \backslash R_1 \backslash^{-1} \bullet \backslash R_2 \backslash^{-1} &= \{[; \ldots; a; \ldots;]\} \bullet \{[; \ldots; b; \ldots;]\} \\ &= \{[; \ldots; a] [;]^k [b; \ldots;] | k \geq 0\} \end{aligned}$$

which is generated by the FE $f_{1,2} = [\varepsilon]^* [a] [\varepsilon]^* [b] [\varepsilon]^*$. Assume now there exists a regular language R_3 s.t. $\backslash R_3 \backslash^{-1} = \backslash R_1 \backslash^{-1} \bullet \backslash R_2 \backslash^{-1}$. But then $R_3 = \backslash(\backslash R_3 \backslash^{-1})\backslash = \backslash(\backslash R_1 \backslash^{-1} \bullet \backslash R_2 \backslash^{-1})\backslash = \{ab\}$, since we can always assume that flattening is surjective, which implies $\backslash(\backslash \backslash^{-1})\backslash = id$, whence $\backslash R_3 \backslash^{-1} = \{[; \ldots; a; \ldots; b; \ldots;], [; \ldots; ab; \ldots;]\}$, which is generated by the FE $f_3 = [\varepsilon]^* ([ab] \cup [a] [\varepsilon]^* [b]) [\varepsilon]^*$ But $L(f_{1,2}) \neq L(f_3)$, because $L(f_3)$ contains the list $[ab] \notin L(f_{1,2})$, which is a contradiction. Clearly the same argument works also for catenation closure, because one sees immediately that the proof does not require that $R_1 \neq R_2$, *i.e.* it works even if $R_1 = R_2 = \{[a]\}$; we omit the detailed proof. □

It is possible to extend the concept of rational transduction to lists. Let $\underline{\tau} : \underline{\Sigma}^{[+]} \to \wp(\underline{\Delta}^{[+]})$ be a rational list transduction, acting through the mapping $\underline{\tau}(\underline{x}) \mapsto \underline{\varphi}(\underline{\psi}^{-1}(\underline{x}) \cap \underline{R})$, for any list $\underline{x} \in \underline{\Sigma}^{[+]}$, where $\underline{R}$ is a RLL; a simple rational list transduction is defined as $\underline{\tau}$, but $\underline{R}$ is a RLL. The adopted definition extends Nivat theorem to lists [Brs79]. From statements 7.7 and 7.4 it follows that RLL and sRLL are closed under rational and simple rational list transduction, respectively. This can also be rephrased by saying that RLL and sRLL are rational list cones; moreover, both are list semi-AFL's since from statement 7.7 RLL and sRLL are union closed. It is also easy to see that both RLL and sRLL are principal cones and principal

semi-AFL's, as regular languages are a principal cone and a principal semi-AFL (any infinite regular language is a generator) and rational transduction, hence list rational transduction, preserves principality.

Classical language theory deals with abstract families of languages: an AFL is a family of languages that is closed under union, catenation closure, intersection with regular languages, direct and reverse homomorphism, and hence also under catenation [Gin75]. We can define an abstract family of list languages as a family of list languages which is a rational list cone and is closed w.r.t. union, list catenation and its closure. From statement 7.8 and the above observations, RLL is an abstract family of list languages (its possible principality is still an open minor problem). Since sRLL is not catenation closed, by statement 7.8, it is not an abstract family of list languages.

Note that flattening maps all the operations: union, list homomorphism and inverse list homomorphism, intersection with regular list languages, list catenation and its closure, onto the corresponding ones in the string domain. Hence the flattened image of a (principal) abstract family of list languages is necessarily a (principal) AFL. This also means that to any rational list transduction $\underline{\tau}$ there corresponds a rational transduction τ s.t. for any list language $\underline{L}$ one has $\backslash\underline{\tau}(\underline{L})\backslash = \tau(\backslash\underline{L}\backslash)$. The word transduction τ is one-to-one associated with the list transduction $\underline{\tau}$ in the obvious way, *i.e.* through the correspondence $\underline{\varphi} \circ (\underline{\psi}^{-1} \cap \underline{R}) \cong \varphi \circ (\psi^{-1} \cap \backslash\underline{R}\backslash)$, if $\underline{\psi}^{-1} \cong \psi^{-1}$, $\underline{R} \cong list(R)$ and $\underline{\varphi} \cong \varphi$.

We extend list merge to the strings of Σ_S^* in the obvious way, e.g. $(ab;\ c)\|(d;\ ef;\ g) = abd;\ ce\,f;g$; its closure is defined accordingly. String merge is associative and admits a neutral element, namely ε, hence its (ε-free) closure is well-defined. String merge can be extended to languages in the obvious way. It would be easy to show that the family of regular languages is closed w.r.t. string merge, by modifying the classical proof of shuffle closure [Eil74].

STATEMENT 7.9 (Merge): *The family FLL is closed w.r.t. merge and merge closure. The family RLL is closed w.r.t. merge, but is not closed w.r.t.* [ ]*-free merge closure. The family sRLL is not closed w.r.t. merge and* [ ]*-free merge closure.*

Proof: For FLL the statement follows from the definition of FE. For RLL, let R_1 and R_2 be any regular languages over the alphabet Σ_S. Then one sees immediately that:

$$list(R_1)\|list(R_2) = list(R_1\|R_2)$$

because the function *list* is one-to-one, hence the merge closure is proved as regular languages are closed w.r.t. string merge.

The family RLL clearly contains finite list languages, which are the languages in $list(\mathcal{F})$, where $\mathcal{F}$ is the family of finite languages. Now, take the finite list language $\underline{L} = \{[a; b]\}$, which therefore is also a RLL. Its [ ]-free merge closure is $\underline{L}^{\|+} = \{[a^n; b^n] | n \geq 1\}$. But $\backslash \underline{L}^{\|+} \backslash = \{[a^n\, b^n] | n \geq 1\}$ is not regular, in contradiction with the fact that every RLL has a regular flattened image, from statement 6.1.

As for sRLL and merge, take as a counterexample the regular languages $R_1 = a^*$ and $R_2 = b^*$. One has $\backslash R_1 \backslash^{-1} = \{[a^*; \ldots; a^*]\}$ and $\backslash R_2 \backslash^{-1} = \{[b^*; \ldots; b^*]\}$, hence $\backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1} = \{[a^*\, b^*; \ldots; a^*\, b^*]\}$. Suppose there exists a regular language R_3 s.t. $\backslash R_3 \backslash^{-1} = \backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1}$. Since flattening can be always assumed to be a surjective function, one has $\backslash(\backslash^{-1}) \equiv id$, hence:

$$\begin{aligned} R_3 &= \backslash(\backslash R_3 \backslash^{-1})\backslash = \backslash(\backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1})\backslash \\ &= \backslash\{[a^*\, b^*; \ldots; a^*\, b^*]\}\backslash = (a^*\, b^*)^* \end{aligned}$$

Therefore, by taking the segmented image of R_3 it follows:

$$\backslash R_3 \backslash^{-1} = \backslash (a^*\, b^*)^* \backslash^{-1} = \{[(a^*\, b^*)^*; \ldots; (a^*\, b^*)^*]\} \neq \backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1}$$

so we have $\backslash R_3 \backslash^{-1} = \backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1}$ by assumption and $\backslash R_3 \backslash^{-1} \neq \backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1}$ as a consequence, which is a contradiction. For instance, the list $[abab; ab] \in \backslash (a^*\, b^*)^* \backslash^{-1}$, yet such list obviously does not belong to $\backslash R_1 \backslash^{-1} \| \backslash R_2 \backslash^{-1}$.

As for sRLL and merge closure, take the finite language $R = \{ab, ba\}$, which then is regular. Its free segmentation $\underline{L} = \backslash R \backslash^{-1} = [\varepsilon]^*\, [a]\, [\varepsilon]^*\, [b]\, [\varepsilon]^* \cup [\varepsilon]^*\, [b]\, [\varepsilon]^*\, [a]\, [\varepsilon]^*$ is a sRLL. Now, it is fairly evident that $\backslash \underline{L}^{\|+} \backslash = Anagrams\ of\ (a^n\, b^n | n \geq 1)$; *see* for instance example 4.3. Were sRLL closed under merge closure, $\underline{L}^{\|+}$ should have a regular flattened image, from statement 6.1, which is not the case, because $Anagrams\ of$ $\{a^n\, b^n | n \geq 1\}$ is evidently not regular (though it is context-free). □

We end the analysis of closures considering mirror, as the traditional families of languages are all mirror closed.

DEFINITION 7.2 (List Mirror): $^R : \underline{\Sigma}^{[*]} \to \underline{\Sigma}^{[*]}$ *is a unary operation on lists onto lists, acting through the mapping:*

$$[x_1; x_2; \ldots; x_n]^R \mapsto [x_n^R; x_{n-1}^R; \ldots; x_1^R] \qquad \text{for} \quad \underline{x} \in \Sigma^{[*]}$$

where $\underline{x} = [x_1; x_2; \ldots; x_n]$ *for* $n \geq 1$, *and posing* $[\ ]^R = [\ ]$.

Equivalently mirror can be defined as $\underline{x}^R \mapsto list\,(string\,(\underline{x})^R)$. Note that $(\underline{x}^R)^R = \underline{x}$ and $\backslash\underline{x}^R\backslash = \backslash\underline{x}\backslash^R$, for any list $\underline{x} \in \underline{\Sigma}^{[*]}$.

STATEMENT 7.10 (Mirror): *The families sRLL and RLL are closed with respect to mirror.*

Proof: The statement follows for sRLL and RLL noting that $(\backslash R_1 \backslash^{-1})^R = \backslash R_1^R \backslash^{-1}$ and that $list\,(R_2)^R = list\,(R_2^R)$, for any regular languages $R_1 \in \Sigma^*$ and $R_2 \in \Sigma_S^*$. □

7.2.4. *Summary of closure properties*

Table 1 shows a summary of the above closure properties of list languages, listing the references of the proofs. Homomorphisms work on lists or strings depending on whether they are applied to list or word languages; list homomorphisms are always associated with word homomorphisms. The symbol $\mathcal{R}$ represents the family of regular languages.

TABLE I
Closure properties of sRLL, RLL, FLL and fFLL.

Closure Properties of List Languages					
	Operation	sRLL	RLL	FLL	fFLL
1a	Hom (alphabetic)	closed (7.4)	closed (7.4)	closed (7.4)	closed (7.4)
1b	Hom (arbitrary)	open (7.4)	closed (7.4)	closed (7.4)	closed (7.4)
2a	Inv. Hom. (alphabetic)	closed (7.4)	closed (7.4)	closed (7.4)	closed (7.4)
2b	Inv. Hom. (arbitrary)	closed (7.4)	closed (7.4)	both are open (*see* above)	
3	Union	closed (7.7)	closed (7.7)	closed (7.7)	closed (7.7)
4a	Intersection with $\mathcal{R}$	undefined	undefined	undefined	open (7.6)
4b	Intersection with sRLL	closed (6.3, 7.7)	closed (7.7)	open (7.6)	undefined
4c	Intersection with RLL	open (6.3, 7.7)	closed (7.7)	open (7.6)	undefined
5	Intersection	closed (7.7)	closed (7.7)	open (7.7)	open (7.7)
6	Complement	closed (7.7)	closed (7.7)	open (7.7)	open (7.7)
7	Catenation	open (7.8)	closed (7.8)	closed (7.8)	closed (7.8)
8	Catenation Closure	open (7.8)	closed (7.8)	closed (7.8)	closed (7.8)
9	Merge	open (7.9)	closed (7.9)	closed (7.9)	undefined
10	Merge Closure	open (7.9)	open (7.9)	closed (7.9)	undefined
11	Mirror	closed (7.10)	closed (7.10)	?	

Closure (2b) fails for FFL and fFLL. In fact, as fFLL contains finite list languages, is union closed, is closed w.r.t. arbitrary direct homomorphism and is catenation closed, were it hypothetically closed also w.r.t. arbitrary reverse homomorphism (this is closure (2b)) by a standard proof [Gin75] it would also be closed w.r.t. the intersection with regular languages; but since such a closure fails to be true (statement 7.6), the hypothesis (closure (2b)) fails, too. The same argument applies also to FLL, with the difference that homomorphisms and catenation apply to lists and that the role of regular languages is played instead by RLL, because the proof contained in [Gin75] depends only on the formal properties of such operations and languages, which are the same for lists and strings; therefore closure (2b) fails also for FLL. Closure (4a) does not make sense for sRLL, RLL and FLL; closures (9) and (10) do not make sense for fFLL. Mirror closure (11) is still an open problem for FLL, hence also for fFLL.

7.3. Decidability properties

We consider now some of the classical decision problems for language families, applied to list languages, examining their solvability. Start by observing that [ ]-freedom and $[\varepsilon]$-freedom are clearly decidable for FLL, hence also for RLL and sRLL (statement 6.3).

STATEMENT 7.11 (Membership): *The membership problems of the families FLL, fFLL, RLL and sRLL are decidable.*

Proof: The proof rests on the construction of an enumeration of the lists generated by the FE in non-decreasing size order. A list $\underline{x}$ *contains* more letters than a list $\underline{y}$ if and only if $|(\backslash \underline{x} \backslash)| > |(\backslash \underline{y} \backslash)|$. There exists an algorithm to list the lists of any FLL, and consequently the strings of any fFLL, in order of increasing number of contained letters and consequently of length, respectively. In fact, it suffices to compute the closure operators in the FE in order of increasing exponents. Let f be a FE, and let $f_i = \alpha_i(f)$, with $i \geq 0$, be the infinite enumerable sequence of FE's obtained by the application of the mappings $\alpha_i : FE \rightarrow FE$, recursively defined as follows:

$$\begin{aligned}
\alpha_i(\underline{x}) &= \underline{x} \quad \underline{x} \in \underline{\Sigma}^{[*]} \\
\alpha_i(\underline{r}_1 \cup \underline{r}_2) &= \alpha_i(\underline{r}_1) \cup \alpha_i(\underline{r}_2) \\
\alpha_i(\underline{r}_1 \| \underline{r}_2) &= \alpha_i(\underline{r}_1) \| \alpha_i(\underline{r}_2) \\
\alpha_i(\underline{r}_1 \bullet \underline{r}_2) &= \alpha_i(\underline{r}_1) \bullet \alpha_i(\underline{r}_2)
\end{aligned}$$

$$\alpha_i(\underline{r}^{\|*}) = \bigcup_{j=0}^{i} \alpha_i(\underline{r})^{\|j} \quad \alpha_i(\underline{r}^{\|+}) = \bigcup_{j=1}^{i} \alpha_i(\underline{r})^{\|j}$$

$$\alpha_i(\underline{r}^{*}) = \bigcup_{j=0}^{i} \alpha_i(\underline{r})^{j} \quad \alpha_i(\underline{r}^{+}) = \bigcup_{j=1}^{i} \alpha_i(\underline{r})^{j}$$

where $\underline{r}$, $\underline{r}_1$ and $\underline{r}_2$ are FE's. Clearly one has that $L(f_{i-1}) \subseteq L(fi)$, for any $i \geq 1$, hence $L(f) = \bigcup_{i=0}^{\infty} L(f_i)$. Now note that any FLL $L(f_i)$ is finite, as it does not contain closures any longer, for any $i \geq 0$, and that any string of $L(f_{i-1})$ does not contain more letters than any string in $L(f_i) - L(f_{i-1})$ does, for any $i \geq 1$; this happens as both f_{i-1} and f_i emulate part of the generative power of f and all the lists generated by f_{i-1} are also generated by f_i, but f_{i-1} iterates fewer times, hence the lists generated by f_i, but not by f_{i-1}, do not contain fewer letters than the ones generated by f_{i-1} alone. Then some standard enumeration procedure of an enumerable family of finite disjoint sets allows to linearly order the lists of $L(f)$ in increasing order of contained letters, allowing to decide the membership problem.

As flattening preserves the number of letters, the same algorithm works also for fFLL. Due to the hierarchy statement 6.3, the membership problem is decidable also for the families RLL and sRLL.

STATEMENT 7.12 (Finiteness and Emptiness): *The emptiness and the finiteness problems are decidable for the families FLL, fFLL, RLL and sRLL.*

Proof: The two problems are simple: a FLL or a fFLL is finite or void if and only if the generating FE does not contain any closure operator or any generator, respectively. Due to the hierarchy statement 6.3 the same problems are decidable for RLL and sRLL. □

STATEMENT 7.13 (Comparison): *The following three comparison problems are undecidable for the families FLL and fFLL, but are decidable for the families RLL and sRLL:*

- *Language equivalence.*
- *Language containment.*
- *Language intersection emptiness.*

Proof: It is shown that these problems for FLL and fFLL encode the same for rational transductions, which are all known to be undecidable [Brs79].

In [Brs79] the following proposition is proved: $\tau : \Sigma^* \to \wp(\Sigma^*)$ is a rational transduction on the free monoid to the family of the subsets of the

free monoid if and only if the set $S = \{(x, y) | \forall x, y \in \Sigma^* \, y \in \tau(x)\}$ is a regular subset of the direct product $\Sigma^* \times \Sigma^*$ of free monoids.

Now, lists of two components can be viewed as elements of the direct product of two free monoids $\Sigma^* \times \Sigma^*$ and their natural piecewise catenation is merge. Hence the above mentioned set S is isomorphic to a FLL, generated by a FE with two-component generators and only using merge. Due to the above proposition, the equivalence, containment and intersection emptiness problems for two such sets are decidable if and only if they are decidable also for some rational transduction. But these problems are known to be undecidable for rational transductions [Brs79], hence also for FLL.

Concerning fFLL, take the FE f_1 that generates $\underline{S} = \{[x; y] | \forall x, y \, (x, y) \in S\}$ and build a new FE $f_2 = f_1 || [\perp;]$, where $\perp \notin \Sigma$; f_2 still uses only merge and two-component generators. The fFLL generated by f_2 is $\{x \perp y | \forall x, y \in \Sigma^* \, y \in \tau(x)\}$; ones sees immediately that $[x; y] \in FLL \Leftrightarrow x \perp y \in fFLL$. Although the separator $(;)$ has been removed by the flattening, it is still encoded in the strings as $\perp$. Hence $\backslash L(f_2) \backslash$ is isomorphic to S and still encodes the above mentioned undecidable problems of rational transductions.

As for RLL and sRLL, taken any RLL or sRLL $\underline{R}$, its word image $string(\underline{R})$ is regular, by definition, and the compaction function *string* is one-to-one (*see* section 2.3). Hence the above comparisons problems (1), (2) and (3) map onto the same problems for regular languages, where all of them are decidable. □

TABLE II
Decision problems for FLL, fFLL, RLL and sRLL.

Decision problems of list languages				
Problem	fFLL	FLL	RLL	sRLL
Membership (7.11)	yes	yes	yes	yes
Emptiness, Finiteness (7.12)	yes	yes	yes	yes
Equivalence (7.13)	no	no	yes	yes
Inclusion (7.13)	no	no	yes	yes
Intersection emptiness (7.13)	no	no	yes	yes

8. COMPARISON WITH THE CHOMSKY HIERARCHY AND AFL

Fair expressions are compared with the Chomsky hierarchy, in order to characterize their power; moreover, some other relations with classical families of languages are proved.

When comparing with word languages, only flattened list languages are considered.

STATEMENT 8.1 (Chomsky Hierarchy): *The family of flattened fair languages (fFLL) is related to the Chomsky hierarchy as follows:*

- *The family fFLL strictly includes regular languages.*
- *The subfamily of the fFLL's generated by $[\varepsilon]$-free FE's is incomparable with context-free languages.*
- *The family fFLL is strictly included in context-sensitive languages.*

Proof: The strict inclusion of the family $\mathcal{R}$ of regular languages in fFLL, *i.e.* $\mathcal{R} \subset fFLL$, follows from $RLL \subset FLL$, proved in statement 6.3, noting that regular languages are the flattened image of RLL; again, the strictness of inclusion follows by the consideration of example 4.1.

Example 4.1 gives a fFLL that is generated by an $[\varepsilon]$-free FE but notoriously is not a context-free language. The language of non-deterministic palindromes $P = \{uu^R | u \in \Sigma^*\}$, over some alphabet $|\Sigma| \geq 2$, is a well-known context-free language. In [Bre93, Fri94] a construction is given for building a one-queue automaton (*see* Cherubini et alii in [Che91] for a review of queue automata and Manna in [Man74] for a related argument, Post machines) that recognizes a given fFLL$\backslash\underline{L}\backslash$. If the flattened FE generating $\underline{L}$ is $[\varepsilon]$-free then the obtained one-queue automaton is quasi-real-time (QRT). But in [Bra88] it is proved that P cannot be recognized by QRT one-queue automata; whence it follows that P is not generated by an $[\varepsilon]$-free FE.

In [Bre93, Fri94] it is proved that any fFLL is recognized by a queue automaton with a single queue tape. A quick inspection of the construction shows that the obtained queue automaton can always be emulated by a linear space bounded Turing machine. The strictness of the inclusion follows from that fFLL's are semilinear, whereas context-sensitive languages are known not to be. □

Notice that the comparisons are made only for flattened list languages, because we have no notion of either a list context-free or a list context-sensitive language.

We conclude this section by listing some partial results that give some insight of the internal structure of the families FLL and fFLL. We start this exploration by examining some simplified types of FE.

STATEMENT 8.2 (One-letter FLL): *Any one-letter fFLL* (i.e. *the alphabet contains only one letter) is a regular language.*

Proof: We have already observed in statement 7.1 that the Parikh image of fFLL coincides with the Parikh image of regular languages. If the FE is one-letter, this identity holds also between fFLL and one-letter regular languages, because both are commutative. □

To proceed we need some classification of simplified FE's, which will be used to define interesting subfamilies of FLL and fFLL: a FE not using catenation closure is said to be *catenation-free*; a FE not using merge closure is said to be *merge-free*.

STATEMENT 8.3 (Catenation-free Subfamily): *The family of the regular languages over a finite direct product of free monoids is a homomorphic image of the subfamily of FLL generated by catenation-free FE. The corresponding subfamily of flattened languages coincides with the family of the homomorphic replications of regular languages* [Gin71].

Proof: A catenation-free FE admits as a homomorphic image a regular expresson over the finite direct product $\times_{k=1}^{n}\Sigma^*$ of free monoids Σ^*, for some $n \geq 1$. In [Brs79] it is proved that the regular languages over the finite direct products of free monoids are isomorphic to the images of multi-tape rational transductions, which in turn by Nivat theorem are isomorphic to the homomorphic replications of regular languages [Brs79, Gin75].

STATEMENT 8.4 (Merge-free Subfamily): *The family of regular languages is a homomorphic image of the subfamily of FLL generated by merge-free FE; the homomorphism is flattening. The corresponding subfamilly of flattened languages coincides with the family of regular languages.*

Proof: Just observe that $\backslash\underline{x}\bullet\underline{y}\backslash = \backslash\underline{x}\backslash\cdot\backslash\underline{y}\backslash$ and $\backslash\underline{x}^*\backslash = \backslash\underline{x}\backslash^*$, for any lists $\underline{x}$, $\underline{y} \in \underline{\Sigma}^{[*]}$, which proves that flattening is the searched homomorphism. Moreover, flattening is a surjective function having as image the whole family of regular languages. □

Recall that FLL also contains a merge-free subfamilly that is isomorphic to regular languages, as already observed in section 3.1: it is the subfamilly of all the FLL's generated by merge-free FE's having generators with only one-letter components.

Given a FE f and two operator instances op_1 and op_2 contained in f, it is said that the instance of op_1 occurs at a *lower* level (in the context of the FE f) than the instance of op_2 if and only if the instance of op_1 appears in an argument of the instance of op_2. Clearly the level relation is an irreflexive partial order of operator instances; op_1 occurs at a *higher* level than op_2

if and only if op_2 occurs at a lower level than op_1. From statement 8.3 it follows immediately a less obvious result.

STATEMENT 8.5 (AFL Hierarchy): *The family of flattened list languages generated by a FE, in which no instance of the merge operator closure occurs at a higher level than any instance of the catenation operator closure, coincides with the rational closure of the family of the homomorphic replications of regular languages.*

Proof: In [Gin71] it is proved that the family of the homomorphic replications of regular languages is a rational cone, hence its rational closure is a well-defined family and is an AFL; such AFL is precisely obtained by taking rational expressions over the generating rational cone. By statement 8.3 a flattened catenation-free FE generates a homomorphic replication of some regular language. String catenation is homomorphic to list catenation and flattening is precisely this homomorphism, hence a FE where no instance of the merge operator closure occurs at a higher level than any instance of the catenation operator closure generates, when flattened, a rational expansion of homomorphic replications of regular languages. The converse is proved in the same way [Brs79]. □

Hence, we have found a non-trivial AFL contained in fFLL. The subfamilies of fFLL listed in statements 8.2, 8.3 and 8.4 are subfamilies of this AFL. Note however that fFLL does not coincide with the rational closure of the homomorphic replications of regular languages, as fFLL is not an AFL. Going back to example 5.1, one sees that the list AntiDyck language $\underline{AntiDyck}$ is generated by a FE where catenation closure occurs at a lower level than merge closure. The language $\underline{AntiDyck}$ is precisely the responsible of the non-closure of fFLL w.r.t. the intersection with regular languages.

We conclude this section by showing inclusion diagrams for the studied families of languages. Figure 4 is an inclusion diagram for the families of regular list languages. Family HRRL is the Homomorphic Replication of Regular Languages and *RatHRLL* is its *Rat*ional closure (*see* Ginsburg in [Gin71]). Families HRRL and *Rat*HRRL are here intended as non-flattened, *i.e.* those subfamilies of FLL s.t. their flattened images are HRRL and *Rat*HRRL, respectively. Figure 5 is an inclusion diagram showing most known facts about the relations of fFLL with other families of (word) languages. Family $fHRRL$ is the Homomorphic Replication of Regular Languages and *RatfHRRL* is its *Rat*ional closure, here considered as flattened

families. Palindromes seem unlikely to be fFLL's (certainly palindromes cannot be flattened images of an $[\varepsilon]$-free fFLL, as they are not QRT one-queue languages; *see* Brandenburg in [Bra88]). The sequences of sample languages $\underline{L}_i$, for $i = 0, 1, 2, 3, 4$ and L_i, for $i = 0, 1, 2, 3, 4, 5$, give some insight of the inner structure of FLL and fFLL, respectively. Such languages can be found in sections 4 and 5, and are also used in section 7.

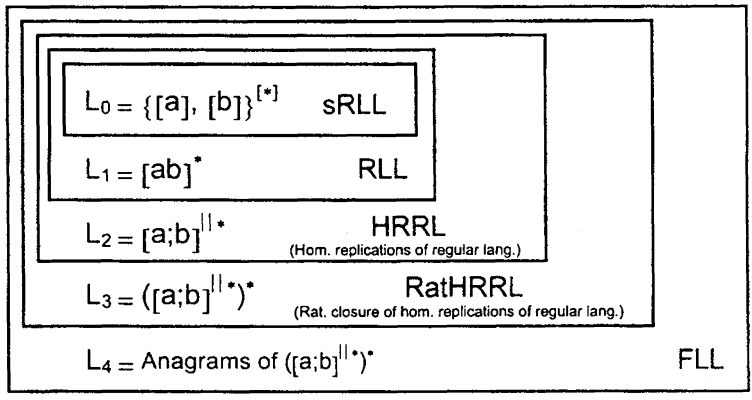

Figure 4. – Inclusion diagram of some subfamilies of FLL.

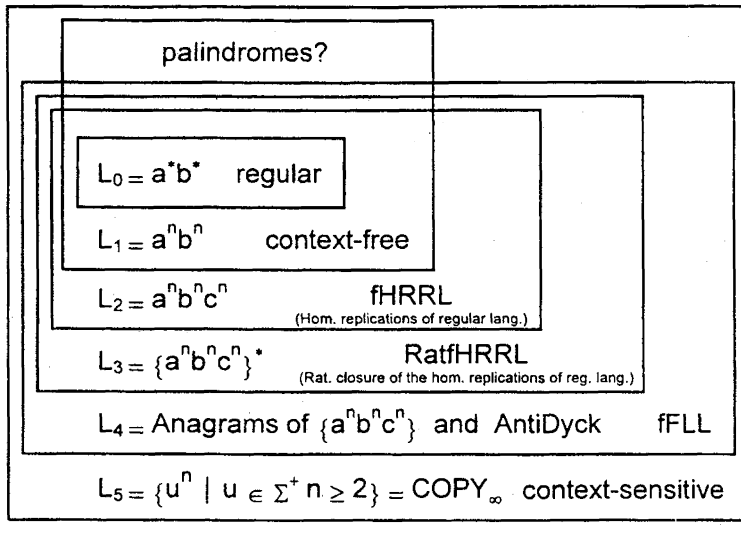

Figure 5. – Inclusion diagram of the family fFLL with related families of languages.

9. ALGEBRAIC RETROSPECTIVES

In this section some connections between fair expressions and other algebraic systems are described, that may enlighten the structure of FE.

9.1. List length

The separator $(;)$ appearing in the lists obeys to rules which are rather different from the ones followed by the lists components.

DEFINITION 9.1 (List Length): *The following function on lists onto the non-negative integers:*

$$\| : \underline{\Sigma}^{[*]} \to N \cup \{0\} \begin{cases} |\underline{x}| = |[x_1; x_2; \ldots; x_n]| \mapsto n & \text{for } \underline{x} \in \underline{\Sigma}^{[+]} \\ \text{and } n \geq 1 \\ |[\ ]| = 0 \end{cases}$$

is named the length of the list $\underline{x}$.

The length of a list clearly coincides with the number of its components.

STATEMENT 9.1 (Length Homomorphism): *The length function is a homomorphism on the structure* $\langle \underline{\Sigma}^{[*]}, \|, \bullet, [\ ] \rangle$ *to the "tropical semiring"* $R = \langle N \cup \{0\}, \max, +, 0 \rangle$.

Proof: In fact, by the definition of length function, one has:

$$|(\underline{x}\|\underline{y})| = \max(|\underline{x}|, |\underline{y}|) \qquad \text{and} \qquad |\underline{x} \bullet \underline{y}| = |\underline{x}| + |\underline{y}|$$

for any lists $\underline{x}, \underline{y} \in \underline{\Sigma}^{[*]}$, and moreover $|[\ ]| = 0$. □

This homomorphism shows that the number of separators $(;)$ in a list follows algebraic rules rather different from those governing the number of letters in the list components.

9.2. List shift

The structure of lists presented in this section can be more algebraically defined as follows: let $\times_{i=1}^{\infty} \Sigma^*$ be the infinite direct product of free monoids equipped with its natural piecewise catenation operator $\cdot$ and a neutral element 1 (which is the element $(\varepsilon, \varepsilon, \ldots)$). Let M be its subset formed by all and only the infinite tuples having only a finite number of non-empty components, *i.e.* different from the empty string ε, and let also $1 \in M$, behaving as a neutral element. Then $\langle M, \cdot, 1 \rangle$ is clearly a monoid.

From now on we identify the tuples of M with finite lists of $\underline{\Sigma}^{[*]}$, by dropping the maximal empty infinite suffix of the tuples of M. Now we can introduce into the monoid M a new operator:

DEFINITION 9.2 (List Shift): *The unary operator of list shift is defined as follows:*

$$s : M \to M \begin{cases} s((x_1; x_2, \ldots, x_n)) \mapsto (\varepsilon, x_1, x_2, \ldots, x_n) & \text{if} \quad n \geq 1 \\ s(1) \mapsto 1 \end{cases}$$

The closure of list shift is defined as follows:

$$s^*(\underline{x}) = \bigcup_{i=0}^{\infty} s^i(\underline{x}) \qquad \begin{cases} s^i(\underline{x}) = \underbrace{s(s(\ldots s(\underline{x})\ldots))}_{i \text{ times}} & \text{if} \quad i \geq 1 \\ s^0(\underline{x}) = \underline{x} \\ s(1) = 1 \end{cases}$$

Any FE expressible in $\langle M, \cdot, s, 1\rangle$ is clearly translatable in $\langle \underline{\Sigma}^{[*]}, \|, \bullet, [\ ]\rangle$, due to the presence in $\underline{\Sigma}^{[*]}$ of the list $[\varepsilon]$, that behaves as the shift operator s, *i.e.* $s(\underline{x}) = [\varepsilon] \bullet \underline{x}$. The converse does not hold, unless one uses a logic language of the second order. In fact, list catenation can only be expressed in terms of shift as follows:

$$\underline{x} \bullet \underline{y} = \underline{x} \cdot s^{|\underline{x}|}(\underline{y})$$

and the operator $s^{|\underline{x}|}$ cannot be expressed in first order logic language.

For instance, $s^{|\underline{x}|}(\underline{y}) = \underline{x}$ is a 2nd order binary predicate; for it is equivalent to asking whether there exist lists $\underline{x}$ and $\underline{y}$ such that by shifting $\underline{y}$ for a number of times equal to the length $|\underline{x}|$ of $\underline{x}$ makes $\underline{y}$ itself equal to $\underline{x}$; since the latter equality (predicate) depends on the former equality (predicate), it defines a 2nd order (binary) predicate.

However, the structure M looks like more natural, from an algebraic point of view, than $\underline{\Sigma}^{[*]}$ and, due to the above observations, any congruence that holds in M is translatable in $\underline{\Sigma}^{[*]}$; equivalently, any congruence that holds in $\underline{\Sigma}^{[*]}$ is a *refinement* of a congruence in M. Thus, the algebraic study of M may reveal details concerning $\underline{\Sigma}^{[*]}$ (this observation is entirely due to Jorge Almeida, of the Universidade de Porto).

Expressions can be defined over the algebraic structure $\langle M, \cdot, s, 1\rangle$ in the same way as FE. To show an example of the utility of this new algebraic structure, consider the following statement:

STATEMENT 9.2 (AntiDyck)1: *The list AntiDyck language* $\underline{AntiDyck}(\Sigma) \subseteq \underline{\Delta}^{[*]}$ *(see* section 5*) is the free language in the algebraic system* $\langle M, \cdot, s, 1\rangle$ *with two-component generators of the type* $(c, \overline{c})$ *as follows:*

$$\underline{AntiDyck}(\Sigma) = \ldots (s^*(\{(c, \overline{c}) | c \in \Sigma\}))^* \ldots ^{\cdots}$$

Proof: The FE f_{11} of example 5.1 generates the list AntiDyck language, and can easily be translated as follows, for $\Sigma = \{a, b\}$:

$$f_{11} = ([\varepsilon]^* [a; \overline{a}] \cup [\varepsilon]^* [b; \overline{b}])^{\|*}$$
$$\Rightarrow \quad (s^*((a, \overline{a})) \cup s^*((b, \overline{b})))^* = (s^*(\{a, \overline{a}), (b, \overline{b})\}))^*$$

Conversely, note that given a generator $(c, \overline{c})$ its free shift s^* yields elements of the type $(\varepsilon, \ldots, \varepsilon, c, \overline{c})$ which are the so-called *generating* lists of example 5.1; then the same considerations as those done in exemple 5.1 apply, and the conclusion is that free combinations of shift and product yield in M all and only AntiDyck lists. □

So there exists a characteristic fair list language $\underline{AntiDyck}$ that admits a very simple and natural generative model in the algebraic system $\langle M, \cdot, s, 1\rangle$.

10. CONCLUSION

We do not know of any closely related formal work on list languages. A reference to an algebraic approach to lists is in Turner [Tur91]. Some vague similarity exists between our model and work on parallel models, in particular Petri nets. The study on concurrent regular expressions by Garg and Ragunath [Gar92] also extends regular expressions with four operators: interleaving, interleaving closure, synchronous composition and renaming (it appears that these four operators are all reducible to compositions of the classical regular ones, plus shuffle and its closure). Concurrent regular expressions are equivalent to Petri nets. Their interleaving or shuffle operator generates any interleaving of two sequences, in contrast to our merge operator which operates on lists instead of strings and orderly interleaves the components of the two lists. Other well-known models of communicating and concurrent systems, in particular Milner CCS [Mil80] and Bergstra "Process Algebra" [Brg85], have used similar operators for interleaving.

The generalisation of the notion of regularity of the list languages is based on the notion of rationality (*i.e.* of fair expressions). We have no notion of recognizability inside FLL, due to the lack of some sort of finite state device for recognizing lists. There also arises the natural question of generalizing the notions of context-freedom, context-sensitivity, etc., to list languages.

On the notion of regular list language we conjecture that any FLL, which is not a RLL, cannot have a flattened image that is a regular language. If proved, this conjecture would state that the family RLL is the largest "regular" subfamily of FLL. We also conjecture that the Parikh function of FLL, also counting the separator (;), is semilinear. If proved, this conjecture would strengthen semilinearity for FLL, showing that the length of the lists of a FLL does not "hide" a non-semilinear behaviour. Another conjecture is that a maximum AFL contained in fFLL exists and is precisely the rational closure of the family of the homomorphic replications of regular languages. If proved, this conjecture would show that the largest "AFL" part of fFLL, *i.e.* the one that enjoys all the traditional closures properties of the commonest language families (e.g. Chomsky type-0,1,2,3 languages, which are all AFL's), is a very simple one, namely the rational closure of HRRL (also an AFL, as proved by Ginsburg in [Gin71]).

Other aspects to be investigated are periodical or "pumping" properties of FLL or fFLL; similar properties have already been proved for a variety of families of queue languages (*see* below), *see* for instance Cherubini *et al.* in [Che9].

An area to be more fully understood concerns automata as recognizers of FLL and fFLL. In [Bre93, Fri94] a proof is shown that the family of flattened fair languages generated by $[\varepsilon]$-free fair expressions is strictly included in the family of languages accepted by deterministic one-queue automata; this is still far, however, from having a complete characterization of recognition for FLL.

Fair expressions do not include flattening as an operator; flattening is just allowed to convert a FLL into a traditional language, whose elements are strings instead of lists. An interesting question is to study the properties of the FE model, enhanced by allowing the use of the flattening operator, with respect to other generative models. In terms of the equivalence of FE with parallel programme schemes [Bre93, Bre94, Fri94], this enhancement can be interpreted as the introduction of modularity in the parallel programme schemes. As for the relations of FE with QRT queue automata [Bre93, Fri94], this enhancement might require passing from one-queue to multiple-queue automata.

A study of generative grammars would complete the picture: how to define generative grammars over lists and to characterize the associated families of languages.

11. ACKNOWLEDGEMENTS

The participation of Alessandra Cherubini and Stefano Crespi Reghizzi to most phases of this research and their help, advices and encouragements are gratefully acknowledged. We express our gratitude also to Jorge Almeida for contributing to and critically revising the algebraic aspects of the work. An anonymous referee has suggested to generalize the Franchi Zannettacci-Vauquelin theorem on the AntiDyck language [Fra80] from word languages to list languages (statement 7.5), and several other improvements as well as has corrected some errors.

REFERENCES

[Ang80] D. ANGLUIN, Finding Patterns common to a Set of Strings, in *Journal of Computer and Systems Sciences*, 1980, *21*, pp. 63-86.

[Brg85] J. A. BERGSTRA and J. W. KLOP, Algebra of communicating Process with Abstraction, in *Theoretical Computer Science*, 1985, *37*, pp. 72-121.

[Bra88] F. BRANDENBURG, On the Intersections of Stacks and Queues, in *Theoretical Computer Science*, 1988, *58*, pp. 69-80.

[Brs79] J. BERSTEL, *Transduction and Context-free Languages*, Teubner Studienbücher, Stuttgart, 1979.

[Bre94] L. BREVEGLIERI , A. CHERUBINI and S. CRESPI REGHIZZI, Fair List Languages and parallel Programme Schemes, in *Developments in Formal Language Theory*, G. ROZENBERG and A. SALOMAA Eds., World Scientific Publishing, 1994, pp. 389-418.

[Bre93] L. BREVEGLIERI, A. CHERUBINI, C. CITRINI and S. CRESPI REGHIZZI, Fair Expressions, Round Robin Concurrency and Queue Automata, Internal Report n° 93-046, Dipartimento di Electronica e Informazione, Politecnico di Milano, Milano, 1993.

[Bre91] L. BREVEGLIERI, A. CHERUBINI and S. CRESPI REGHIZZI, Quasi-Real-Time Scheduling by Queue Automata, in *Lecture Notes in Computer Science*, J. VYTOPIL Ed., Springer-Verlag, 1991, 571, pp. 131-147.

[Che91] A. CHERUBINI, C. CITRINI, S. CRESPI REGHIZZI and D. MANDRIOLI, QRT FIFO Automata, Breadth-first Grammars and their Relations, in *Theoretical Computer Science*, 1991, 85, pp. 171-203.

[Eil74] S. EILENBERG, Automata, Languages and Machines, vol. A, Academic Press, 1974.

[Fra80] P. FRANCHI ZANNETTACCI and B. VAUQUELIN, Automates à File, in French, Queue Automata, in *Theoretical Computer Science*, 1980, *11*, pp. 221-225.

[Fri94] M. FRIGERIO, Espressioni Fair, Processi paralleli e Automi a Coda, in Italian, Fair Expressions, parallel Processes and Queue Automata, Thesis, Università degli Studi di Milano, Faculty of Information Sciences, Milano, Italy, 1994-1995.

[Gar92] V. K. GARG and M. T. RAGUNATH, Concurrent regular Expressions and their Relationship to Petri Nets, in *Theoretical Computer Science*, 1992, *96*, pp. 285-304

[Gin75] S. GINSBURG, Algebraic and Automata-theoretic Properties of formal Languages, North Holland, 1975.

[Gin71] S. GINSBURG and E. H. SPANIER, AFL's with the semilinear Property, in *Journal of Computer and Systems Sciences*, 1971, *5*, pp. 365-369.

[Har78] M. HARRISON, *Introduction to formal Languages*, Addison Wesley, 1978.

[Man74] Z. Manna, *The mathematical Theory of Computation*, McGraw Hill, 1974.

[Mil80] G. J. MILNER, A Calculus of communicating Systems, in *Lecture Notes in Computer Science*, Springer-Verlag, 1980, *92*.

[Sal73] A. SALOMAA, *Formal Languages*, Advanced Computing Machines Monograph Series, Academic Press, 1973.

[Tur91] D. TURNER, Duality and De Morgan Laws for the Algebra of Lists, in *Bulletin of the European Association of Theoretical Computer Science*, 1991, *45*, pp. 229-237.