CONCISENESS OF ASSOCIATIVE LANGUAGE DESCRIPTIONS[1]

STEFANO CRESPI REGHIZZI†, MATTEO PRADELLA, PIERLUIGI SAN PIETRO
Politecnico di Milano
Dipartimento di Elettronica e Informazione
P.za L. da Vinci 32, I-20133 Milano
e-mail: {crespi|pradella|sanpietro}@elet.polimi.it
†*Università della Svizzera Italiana,*
Ist. di Tecnologie della Comunicazione

ABSTRACT

Associative Language Descriptions are a recent grammar model, theoretically less powerful than Context Free grammars, but adequate for describing the syntax of programming languages. ALD do not use nonterminal symbols, but rely on permissible contexts for specifying valid syntax trees. In order to assess ALD adequacy, we analyze the descriptional complexity of structurally equivalent CF and ALD grammars, finding comparable measures. The compression obtained using CF copy rules is matched by context inheritance in ALD. The family of hierarchical parentheses languages, an abstract paradigm of HTML, and of expressions with operator precedences are studied in detail. A complete ALD grammar of Pascal is presented to testify of the practicality of the ALD approach.

Keywords: Context Free Grammars, Associative Grammars, Grammar Size, Context Inheritance, Descriptional Complexity.

1. Introduction

In spite of their widespread use in computer science, context-free (CF) grammars have shortcomings that prompted the search of alternative models for defining languages. The recently proposed [2] Associative Language Description (ALD) model combines two historically prominent approaches of formal linguistics: the structural Chomskian method and the associative (or distributional) method. From the former ALD takes the tree structure of CF rules. From the latter ALD imports the notion of local testability and context association. An ALD consists of a set of purely terminal patterns accompanied by contextual constraints; it therefore qualifies as a "pure" grammar formalism in the sense of [5], since it does not use nonterminal (n.t.) symbols. Also Marcus' contextual grammars [7], the best known pure grammar formalism (i.e., n.t.-free), have been motivated by a similar interest for associative linguistic approaches.

The ALD language family is strictly smaller than the CF family, since it is included in the Non Counting (NC) CF family of [4]. Yet this reduction of generative capacity does not defeat ALD descriptional adequacy, because the CF ability to perform modulo counting is never used in artificial or human languages: in other words NC is a linguistic universal. A confirmation comes from the finding that Greibach's hardest context-free language belongs to the ALD family [2].

Incidentally, the ALD investigation was initially motivated by the search of a language model that would be consistent with modern brain theory [3], but the present work does not go in that direction, and considers computer languages. A new proposal should be defended by showing its advantages over consolidated models, a challenge we are taking here and in related investigations.

In this paper, we provide some evidence of ALD usability for practical language definition, by comparing the descriptional complexity of CF and ALD grammars. The exposition starts in

[1]Work supported by MURST- Cofinanziamento 9801204372/2 1999 and by CNR-CESTIA.

Section 2 with the basic definitions and properties of ALD, in particular that an ALD admits a structurally equivalent CF grammar. Simple bounds on the descriptional complexity of the two models are then derived in Section 3. Section 4 considers some important varieties of multi-level structures that occur in numerous technical languages (Mark-up languages such as HTML, expressions with operator precedence, etc.), and analyses and compares their descriptional complexity using ALD and CF rules. It also observes that there is a duality between the combinatorial growth of ALD permissible contexts and the growth of CF productions in copy-free grammars and that a notation called context inheritance may reduce the complexity by the same amount as the introduction of the CF copy rules. Finally, Section 5 exhibits a large piece of empirical evidence, by comparing the ALD and CF grammars of the programming language Pascal. In the conclusions, other relevant aspects of ALD theory are mentioned.

2. Basic definitions

We use a special character $\perp$, not present in the terminal alphabet Σ, called the left/right *terminator*, which encloses the sentences of the language. Next, we define the tree structures that are relevant for ALD. Let Δ, the *place holder*, be a new character.
A tree whose internal nodes are labelled by Δ and whose leaves are in Σ is called a *stencil tree*. Such a tree is composed by juxtaposed subtrees having height one and leaves in $\Sigma \cup \{\Delta\}$, called *constituents*. The *frontier* of a stencil tree T is denoted by $\tau(T)$, while $\tau(K_i)$ denotes the frontier of the constituent K_i.

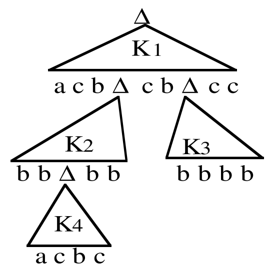

Figure 1: A stencil tree with four constituents schematized by triangles

Definition 1 *For an internal node i of a stencil tree T, labelled by Δ, let K_i and T_i be resp. the constituent and the maximal subtree of T having root i. Introduce a new symbol $k_i \notin \Sigma \cup \{\perp, \Delta\}$, and consider the tree T_{K_i} obtained replacing in T the subtree T_i with k_i. Consider the frontier of T_{K_i}, $\tau(T_{K_i}) = sk_it$ with $s, t \in \Sigma^*$. The strings $\perp s$ and $t\perp$ are called, resp., the* left *and* right contexts *of K_i in T and of T_i in T: $left(K_i,T)=left(T_i,T)=\perp s$ and $right(K_i,T)=right(T_i,T)=t\perp$.*
For instance, in Fig. 1 the left context of K_3 is $\perp$acbbbacbcbbcb and the right context of K_1 is $\perp$: the terminators are automatically prefixed/appended. Next comes the main definition.

Definition 2 *An* Associative Language Description (ALD) *A consists of a finite collection of triples $<x,z,y>$ or* rules, *(also written x[z]y), where $x \in (\perp\Sigma^*) \cup \Sigma^+$, $y \in (\Sigma^*\perp) \cup \Sigma^+$, and $z \in (\Sigma \cup \{\Delta\})^*$. For a rule, the string z is called the* pattern *and the strings x and y are called the* permissible left/right contexts.

In order to make ALD specifications more concise we also adopt the following short notations:
Two (or more rules) $x_1[z]y_1$, $x_2[z]y_2$ can be merged into the complex rule: $x_1,x_2[z]y_1,y_2$, which can be merged with $x_1,x_2[w]y_1,y_2$ into the rule $x_1,x_2[z][w]y_1,y_2$.
Moreover, if a left/right permissible context is irrelevant, it can be replaced by the "don't care" symbol '-' or ε. Finally, we introduce the new symbol Λ instead of $(\Delta \cup \varepsilon)$.
An ALD provides a set of test conditions for checking the validity of a stencil tree, namely that the actual contexts of each constituent is consistent with the permissible contexts of some rule.

Definition 3 *A tree T is* valid for an ALD A *iff for each constituent K_i of T, there exists in A a rule $u[\tau(K_i)]v$ where u is a suffix of the left context of K_i in T and v is a prefix of the right context of K_i in T. In that case we say that such a rule matches the context of K_i in T.*
Therefore, an ALD is a device for defining a set of stencil trees and a string language, corresponding to their frontiers.

Definition 4 *The* (stencil) tree language *$T_L(A)$ defined by an ALD A is the set of all stencil trees valid for A. The* (string) language *defined by the ALD A, denoted by L(A) is the set of the strings $x \in \Sigma^*$ such that $x = \tau(T)$ for some tree $T \in T_L(A)$. Two ALDs are* structurally equivalent *if they define the same tree language; they are* equivalent *if they define the same (string) language.*

Example 1 The language $\{a^n cb^n \mid n \geq 1\}$ is defined by the ALD rules: $\perp[a\Delta b]\perp$, $a[a\Delta b]b$, $a[c]b$.
Both contexts can be dropped from the first two rules, and the right (or left but not both) context can be omitted from the last rule, obtaining the equivalent description: $-[a\Delta b]-$, $a[c]-$
The Dyck language is defined by the ALD rules: $-[b\Delta e\Delta]-$, $-[b\Delta e]-$, $-[be\Delta]-$, $-[be]-$, $-[\varepsilon]-$
where all contexts are "don't care", which can be written more concisely as $-[b\Lambda e\Lambda]-$, $-[\varepsilon]-$.
The phenomenon of *ambiguity* can occur in ALDs much as in CF grammars. For example, the following rules ambiguously define the Dyck language: $-[\Delta\Delta]-$, $-[b\Delta e]-$, $-[\varepsilon]-$, because a sentence like 'bebe' admits distinct tree structures.

Definition 5 *An ALD is* reduced *if each rule is used, i.e. matches some constituent, in some valid stencil tree.*

Given an ALD A and a rule $<x,z,y> \in A$ introduce some integer attributes:

- the *left* (resp. *right*) *degree* of a rule is |x| (resp. |y|), for '-' the length is considered to be zero;
- the *degree* of a rule is $\max\{|x|, |y|\}$;
- the *degree* of an ALD is the maximum of the degrees of its rules;
- the *width* of an ALD is the maximum of the length of its patterns.

A tree language T is of degree (resp. width) k, if there exists an ALD A of degree (resp. width) k such that $T=T_L(A)$; similarly for a string language. Width and degree induce two infinite hierarchies of ALD language families [2]

Definition 6 *An ALD A of degree k is* homogeneous *iff:*

$$A \subseteq (\{\Sigma^k \cup \perp\Sigma^j \mid 0 \leq j < k\}) \times (\Sigma \cup \Delta)^* \times (\{\Sigma^k \cup \Sigma^j\perp \mid 0 \leq j < k\})$$

Theorem 1 *For every ALD A there exists a structurally equivalent homogeneous reduced ALD A'.*

Sketch of the Proof. Let k be the degree of A. A homogeneous structurally equivalent ALD is A" $= \{<ux,z,yv> \mid <x,y,z> \in A \wedge x, y \in \Sigma^* \wedge u \in \Sigma^{k-|x|} \wedge v \in \Sigma^{k-|y|}\}$; this is not necessarily reduced because it may contain unusable contexts. Let $C_k \subseteq \bigcup_{0 \leq i,j<k} (\perp\Sigma^i \times \Sigma^j\perp) \cup (\Sigma^k \times \Sigma^k)$ be the set of all the contexts of all the constituents of all the valid trees of A. By definition, A' is $\{<u,p,v> \mid <u,v> \in C_k \wedge <u,p,v> \in A''\}$. □

Theorem 2 *If A' and A" are two structurally equivalent reduced homogeneous ALDs, then A' =A".*

This immediate property is needed later and is in contrast with CF grammars where two structurally equivalent grammars may be different.

3. The complexity of ALD and CF descriptions

For a meaningful comparison of the ALD and CF models, we focus our attention on structurally equivalent grammars, because it is generally accepted both in linguistics and in compiler technology that a grammar should be structurally adequate.
In the following, let G = (V, Σ, P, S) be a CF grammar, where V are the n.t.s, P the productions and S the axiom. We assume the parentheses '(' and ')' not to be in Σ.

Definition 1 *The* CF parenthesis grammar *associated with the CF grammar G is* $G' = (V, \Sigma\cup\{(,)\}, P', S)$ *where* $P' = \{B \to (\alpha) \mid B \to \alpha \in P \wedge \alpha \notin V\} \cup \{B \to C \mid B \to C \in P \wedge C \in V\}$. *Likewise, the parenthesis ALD grammar A' associated with an ALD grammar A is the set:* $\{<x,z,y> \mid z = (p) \wedge <x,p,y> \in A\}$.
The parenthesis language *of the CF grammar G or of the ALD grammar A is the language generated by G' or A', respectively. Two grammars are* structurally equivalent *iff they generate the same parenthesis language. The rules* $B\to C$, $C\in V$ *are called* copy rules. *A CF grammar is* copy-free *if no production in P is a copy rule.*

It is well-known that any CF grammar admits a structurally equivalent copy-free grammar. Structural equivalence, as defined here, disregards the language structure created by copy rules, since the latter are viewed as a mere expedient for reducing grammar size. We recall from [2] the next fundamental property.

Theorem 1 *For any ALD there exists a structurally equivalent CF grammar.*

The proof is based on the idea to categorize the ALD constituents subtrees by syntax categories characterized by four parameters: <*le, fi, la, ri*>, where *le* and *ri* are resp. the left and the right context of the subtree; *fi* and *la* are the first and the last k terminal characters of τ(T) (see Definition 1), where k is the degree of the ALD.
Observing that the n.t.s of the equivalent CF grammar are 4-tuples <*le, fi, la, ri*>, the next statement follows immediately.

Corollary 2 *For any ALD of degree k, there exists a structurally equivalent CF grammar having at most* Σ^{4k} *n.t.s.*

This bound is too pessimistic; however, instead of refining it we reverse the comparison, by relating the size of a given CF to the size of the structurally equivalent ALD, when the latter exists. We denote G' the parenthesis grammar associated with a CF grammar G (see[8]).

Definition 2 *Let* $V_s = \Sigma \cup V \cup \{(,)\}$. *A string* $\beta \in V_s{}^*\Delta V_s{}^*$, *where* Δ *is the place holder, is called a* McN-pattern[2]. *Moreover, for* $B \in V$ *denote with* $\beta[B]$ *the result of the replacement of the placeholder in* β *with B.*
Two n.t.s $B, C \in V$ *are* equivalent *iff* $\forall\beta\ (S \to^+ \beta[B] \Leftrightarrow S \to^+ \beta[C])$. *A n.t. B is useless iff there exists no* β *such that* $S \to^+ \beta[B]$ *or there exists no* $t \in \Sigma^*$ *such that* $B \to^+ t$.
A CF grammar is reduced *iff it does not contain equivalent or useless n.t.s.*

It is well-known that for any CF language there exists an equivalent reduced grammar.
Next, we consider the unbounded and bounded contexts of a n.t. and a mapping from the right-hand sides of CF productions to the patterns of the ALD rules.

[2] It corresponds to the definition of pattern in [8], which has nothing to do with the "pattern" of ALD.

Definition 3 *For a n.t. B of a CF grammar* $G = (V, \Sigma, P, S)$ *the* context of B *is* $Con(B) = \{<u,v> \mid u, v \in \Sigma^* \wedge S \to^+ uBv\}$
The context of degree $k \geq 1$ *(or* k-context*) is* $Con_k(B) = \{<u',v'> \mid u' \in \Sigma^k \cup \bot\Sigma^j,\ v' \in \Sigma^k \cup \Sigma^j\bot$, *with* $0 \leq j < k$; $<u,v> \in Con(B)$, *u' is a suffix of u, v' is a prefix of v*$\}$

Definition 4 *Let* $h : (V \cup \Sigma)^* \to (\{\Delta\} \cup \Sigma)^*$ *be an alphabetic homomorphism defined as* $h(B) = \Delta$ *for* $B \in V$, $h(v) = v$ *for* $v \in \Sigma$. *For a n.t.* $B \in V$ *of G, the* pattern set of B *is defined as* $Pat(B) = \{h(\alpha) \mid B \to \alpha \in P \wedge \alpha \notin V\}$.

The condition $\alpha \notin V$ is to disregard the copy-rules of G, which otherwise would generate the useless pattern $[\Delta]$.
Let $G = (V, \Sigma, P, S)$ be a reduced CF grammar, structurally equivalent to an ALD A of degree k. The next statements hold.

Theorem 3 *(i) The set of patterns of A is* $\cup_{B\in V} Pat(B)$; *(ii) for any copy rule* $B \to C$, *consider* $B \to \beta$ *and* $C \to \gamma$ *such that* $\beta, \gamma \notin V$. *Then A includes the rules* $x[h(\beta)]y$ *and* $v[h(\gamma)]w$ *such that* $<x,y> \in Con_k(B)$, $<v,w> \in Con_k(C)$, *and the permissible contexts of* $h(\gamma)$ *are included in the permissible contexts of* $h(\beta)$.

Proof. (i) Since G is reduced and A is its structurally equivalent ALD, every $h(\alpha)$, with $D \to \alpha \in P$, is required as pattern of A. By definiton of ALD (Definition 3, in Section 2), a pattern $[\Delta]$ is useless. Therefore, the set of patterns of A is $\{h(\alpha) \mid D \in V \wedge D \to \alpha \in P \wedge \alpha \notin V\}$ which is equal to $\cup_{D \in V} Pat(D)$. (ii) First part: Since G is reduced, $h(\beta)$ and $h(\gamma)$ are required as patterns of A. Second part: the derivations $S \to^* B \to \beta$ and $S \to^* B \to C \to \gamma$ prove the inclusion. □

Therefore, the structurally equivalent ALD is $A=\{x[z]y \mid B\in V \wedge <x,y>\in Con_k(B) \wedge z\in Pat(B)\}$, to be also abbreviated as $A = \{<Con_k(B), Pat(B)> \mid B \in V\}$.
We can now state a first comparison of the relative size of ALD and CF grammars, which follows immediately from the definition of the homomorphism h and of the set Pat(B), $B \in V$:

Theorem 4 *For every n.t.* $B \in V$, $|Pat(B)| \leq |\{\alpha \mid B \to \alpha \in P\}|$.

Therefore, the number of distinct patterns of A is less than or equal to the number of productions of G. It is equally important to examine the number of permissible contexts of the ALD.

Theorem 5 *The cardinality of the set* $\{Con_k(B) \mid B \in V\}$ *(*distinct permissible context sets*) of the ALD A is* $|V|$.

Sketch of the Proof. Let $B, C \in V$. It is known ([8]) that B is equivalent to C iff $Con(B) = Con(C)$. From the existence of the k-degree ALD A structurally equivalent to G, it descends that $Con(B) = Con(C)$ iff $Con_k(B) = Con_k(C)$, because the contexts can be truncated to the k characters preceding and following B and C.
Since G is a reduced there are no equivalent n.t.s B and C, therefore we have: $\forall B, C \in V\ (B \neq C \Rightarrow Con_k(B) \neq Con_k(C))$, which is our statement. □

4. Comparisons for some ubiquitous syntax structures

In what follows we apply the previous results to a typical structure that occurs, disguised under different concrete syntaxes, in many artificial languages. We refine the results by counting the number of symbols in both CF and ALD descriptions.
Actually, at the symbol level the dimensions of grammars depend very much on the chosen representation, since various compression techniques can be used. For example, every language

designer knows that copy rules are a very effective method for reducing the size of CF grammars, especially when they feature some sort of hierarchical structure. We will apply an analogous technique for the ALD, based on the concept of *context-inheritance*.
Another useful technique, to be extensively used in the Pascal description of Sect. 5, is to express the permissible ALD contexts as the cartesian product of their left and right components.
The next definitions [10] are needed to compare CF and ALD sizes.

Definition 1 *The* size $|H|$ *of a CF/ALD grammar H is the sum of the lengths of their rules, considering only the symbols in* $Sym = V \cup \Sigma \cup \{\varepsilon\}$ *for the CF case and in* $Sym = \{\Delta, \Lambda, \perp, \uparrow, \varepsilon\} \cup \Sigma$ *for the ALD case. The* norm *of a CF/ALD grammar H is defined by* $||H|| = |H| \log |Sym|$.

The size expresses the grammar length in symbols, while the norm expresses its length in bits. The bit level measures in our case are no more informative than the symbol level measures, because in our examples |V| is always proportional to |Σ|, and the CF grammar alphabet uses at most |V| more symbols than the ALD grammar. This difference appears as an argument of the logarithm: $\log |V \cup \Sigma|$ is always $O(\log(|V|))$. Therefore from now on we will only compare the *size* of CF/ALD grammars.
The abstract syntax structures we want to compare occur in different incarnations such as:

- arithmetical (or Boolean) expressions with operators ranked into n precedence levels; examples occur in predicate calculus and in the arithmetical part of any programming language.
- parenthesis structures featuring n different pairs of parenthesis, ranked into a hierarchy. The typical example is HTML, an instance of the generalised mark-up languages SGML.

To focus on one concrete case, we consider now the language with n≥1 pairs of parentheses ranked by level, in which a parenthesis pair of level i may only contain parentheses of lower levels. The terminal alphabet is $\Sigma = \{<_1, >_1, ..., <_n, >_n\}$. An instance of a valid string is:

$<_1 >_1 <_5 <_2 >_2 <_3 <_1 >_1 <_1 >_1 >_3 <_1 >_1 >_5 <_3 >_3$

The ALD grammar of degree 1 is:

ALp: - $[\ \varepsilon\ ] \perp$
- $[<_1 >_1 \Delta] \perp, >_2, ..., >_n$
- $[<_i \Lambda >_i \Delta] \perp, >_{i+1}, ..., >_n$ where $1 < i \leq n$.

The size of ALp is $|ALp| = 1/2\ n^2 + 9/2\ n + 1$.
The contexts are included one in the other, from top to bottom. This hierarchy of contexts can be concisely represented using *context-inheritance*, encoded by a new character ↑.

Definition 2 *A* rule block *is a triple* $<X,Z,Y>$, *where* $Z \subseteq (\Sigma\cup\{\Delta\})^*$ *and (X,Y) is a sequence* $(x_1, y_1), (x_2, y_2), ..., (x_j, y_j)$, *for some* $j > 0$, *such that* $\forall h,\ x_h \in \{\uparrow\}\cup(\perp\Sigma^*)\cup\Sigma^+,\ y_h \in \{\uparrow\}\cup(\Sigma^*\perp)\cup\Sigma^+$.
An ALD with inheritance *is a sequence of rule blocks.*

For every ALD with inheritance H there exists a structurally equivalent ALD A so defined: for every pair of rule blocks <L, P, R>, <M, Q, S>, which are adjacent in the sequence H, the following set of rules are in A:

$\{<m_i, q, s_i> \mid m_i \in M \wedge s_i \in S \wedge q \in Q \wedge m_i \neq \uparrow \wedge s_i \neq \uparrow\} \cup$
$\{<l, q, s_i> \mid l \in L \wedge s_i \in S \wedge q \in Q \wedge m_i = \uparrow \wedge s_i \neq \uparrow\} \cup$
$\{<m_i, q, r> \mid m_i \in M \wedge r \in R \wedge q \in Q \wedge m_i \neq \uparrow \wedge s_i = \uparrow\} \cup$
$\{<l_i, q, r_i> \mid l_i \in L \wedge r_i \in R \wedge q \in Q \wedge m_i = \uparrow \wedge s_i = \uparrow\}$

No other rule is in A.
Using inheritance the ALp becomes:

AILp: - $[\ \varepsilon\ ][<_n \Lambda >_n \Delta] \perp$
- $[<_{n-i} \Lambda >_{n-i} \Delta] \uparrow, >_{n-i+1}$ where $i \in 1..n-1$.

It is straightforward to note that the size of this grammar is linear with respect to n: $|AILp| = 6n$.
In our opinion the ALD with inheritance more neatly expresses the hierarchy of the language.

We now compare AILp with different CF grammar versions for the same language. First, we consider a structurally equivalent CF grammar, *reduced, invertible* and *copy-free*.

GMLp: $N_1 \to <_1 >_1 N_1 \mid \varepsilon$

$N_i \to <_i N_j >_i N_k$ where $i \in 2..n; j \in 1..i\text{-}1; k \in 1..i$

and the axioms are $\{N_i \mid i \in 1...n\}$.

Obviously, GMLp is invertible because the right parts of the rules are pairwise distinct.

To see that GMLp is reduced, consider the set of derivations $\{N_1 \to <_1 >_1\} \cup \{N_i \to <_i N_1 >_i \to <_i >_i \mid 2 \le i \le n\}$: every n.t. derives some terminal string. Consider, then, a McN-pattern such as $\beta = <_2 \Delta >_2$. It is possible to derive, from some axiom S, $S \to <_2 N_1 >_2$. On the other hand, it is impossible to derive from an axiom any element of the set $\{<_2 N_j >_2 \mid j \in 2...n\}$: N_1 is distinguishable from the other n.t.s. Similarly N_2 is distinguishable from any N_j, $j \in 3...n$, using the McN-pattern $\beta = <_3 \Delta >_3$. Similarly we distinguish each N_j, $j \in 3...n$, from the others.

The size of this grammar is: $|GMLp| = 4/3\ n^3 - 1/3\ n + 4$. McNaughton's minimality theorem [8] states that the grammar has the minimum number of n.t.s, with respect to the class of invertible, structurally equivalent grammars. However, we can easily design a more compact structurally equivalent grammar, dropping the invertibility hypothesis.

GLp: $N_i \to <_i N_{i-1} >_i N_i \mid <_{i-1} N_{i-2} >_{i-1} N_{i-1} \mid .. \mid <_1 >_1 N_1 \mid \varepsilon$, $i \in 1...n$, N_n is the axiom.

The size of this grammar is $|GLp| = 2n^2 + 3n$.

This size is comparable with the size of the first ALD grammar, ALp.

We can build an even smaller CF grammar for the same language, using copy rules.

GCLp: $N_1 \to <_1 >_1 N_1 \mid \varepsilon$

$N_i \to <_i N_{i-1} >_i N_i \mid N_{i-1}$, where $i \in 2...n$.

The size is $|GCLp| = 6n - 1$, quite comparable to the ALD with inheritance.

The above quantitative results hold for other concrete incarnations of the same abstract syntax paradigm. A ubiquitous case is provided by expressions with operators, ranked by precedence, such as +, × in the case of arithmetic expressions. We also admit one type of parenthesis.

Consider the *generalized expression language* with n levels of operators (denoted by $\#_1, \dots, \#_n$), defined by the ALD with inheritance:

AILe: $(, \perp [\ \Delta\ \#_n\ \Delta\ ]$ -

$\uparrow, \#_n [\ \Delta\ \#_{n-1}\ \Delta\ ]$ - ... $\uparrow, \#_2 [\ \Delta\ \#_1\ a\ ][\ a\ ][(\ \Delta\ )]$ -

Clearly, its abstract structure is essentially the same of ALp. Analogously, a structurally equivalent CF grammar with copy rules is the following:

GCLe: $N_1 \to N_1\ \#_1\ a \mid a \mid (N_n)$

$N_i \to N_i\ \#_i\ N_{i-1} \mid N_{i-1}$; where $i \in 2..n$

and its structure is almost the same of GCLp.

The size comparisons between the various CF and ALD grammars are collected in the next table.

	Hierarchical Parentheses		**Expressions with operator precedence**	
	Context Free	ALD	Context Free	ALD
CF copy-free / ALD	$2n^2+3n$	$1/2n^2+9/2n+1$	$3/2n^2+13/2n$	$1/2n^2+9/2n+4$
CF copy rules / ALD inher.	$6n-1$	$6n$	$5n+3$	$5n + 4$
CF copy-free & invertible	$4/3n^3-1/3n+4$	-	n^3+7	-

In conclusion, the descriptional complexity of languages with hierarchically nested constructs is the same for CF and ALD representations, and both benefit from the use of sub-categorization, be it either in the form of copy rules or of context inheritance.

5. Associative Language Description of Pascal

In order to assess in practice the descriptive capacity of the ALD model, and to compare its convenience with the established CF model, we have worked on two real languages, HTML and Pascal. We report on the latter, since the former has simpler structures already considered in Sect.

4. It was rather surprising to discover that Pascal is an ALD language, meaning by that that the same approximation provided by a BNF grammar can be obtained using an ALD (semantic constraints exceed the descriptive capacity of both models). The Appendix lists a complete ALD grammar of Pascal.
The maximum degree 3 is required in one localized construct of Pascal, the specification of parameters of a function or procedures. The permissible left contexts are the two strings "function id (", "procedure id (" and the right context is "don't care". The rather long left context is needed to distinguish the construct (formal parameters declaration):

"function id (", "procedure id ("	[Δ ;] [var] Δ : Δ id	

from the construct (actual parametrs list):

~{function, procedure} id "(" ∪ {",", "id ["}	Δ, Δ	")","]"

The last context description expression is based on negative examples. In fact *~{function, procedure} id "("* means all the strings *a id(*, where $a \in \Sigma$ is a token different from *function* or *procedure*. Sometimes it is more economical to specify the forbidden left/right context, instead of the permissible ones. This is analogous to the description of exceptions, instead of regularities.
For most rules, the degree one suffices, with a majority of rules having don't cares in at least one context. For example, the context <with, ->, used in the third rule, stands for the set of contexts $\{<u.with, v> \mid u \in \Sigma^2, v \in \Sigma^3\}$.
The ALD takes advantage of two combined techniques for improving readability and reducing size. Context inheritance factorizes repeated context sets of arithmetic or Boolean expressions as explained in Sect. 4. Cartesian products of left and right context sets are extensively exploited, instead of sets of pairs (left context, right context). This simple but effective technique is described next.
Consider a CF grammar as in theorem 2, section 3, and its structurally equivalent ALD of degree k. Let $L(\mathrm{Con}_k(B))$ (resp. $R(\mathrm{Con}_k(B))$) denote the projection of $\mathrm{Con}_k(B)$ on the left (resp. right) component.

Theorem 1 *Consider a n.t. B and the corresponding ALD rule $<Con_k(B), Pat(B)>$. If for every n.t. $C \neq B$ the following condition holds*

$$Con_k(B) \cap Con_k(C) = (L(Con_k(B)) \times R(Con_k(B))) \cap Con_k(C),$$

then the rule $<Con_k(B), Pat(B)>$ can be replaced by $<L(Con_k(B)), Pat(B), R(Con_k(B))>$, i.e., the permissible contexts of Pat(B) are all the pairs in the Cartesian product.

Notice that the conditions prevent the unwanted interchange between B and C. To illustrate, consider the definitions of the array, file and set constructs of Pascal in the Appendix.
A general remark on the structure of ALD definitions is that the patterns represent the shortest valid strings that may occur for the various constructs of the language. The patterns in general include one or more place-holders, that permit to increase the construct by insertions of other patterns. Insertions are of course controlled by context permissibility.
A shortcut device that we have not used in this ALD is the lambda notation introduced earlier for optional place-holders. Its use would permit to drop the epsilon patterns.
We have compared the ALD with the CF grammar of Pascal, that can be found in many Pascal textbooks and we found the sizes to be about the same. Although the presented ALD version already improves on our first attempts, it is not claimed to be the best possible, especially because of the great difference in maturity and experiences between ALD and CF modeling.

6 Conclusions

Other compression techniques for ALD have been examined but have not been investigated in depth. Multiple inheritance of contexts is another idea that we have found to be effective in certain parts of Pascal. It consists in importing parts of the context from different rules. To implement multiple inheritance, rules must be numbered and the inheritance indicator (↑) must be

qualified with the reference to the intended ancestor. In our ALD for Pascal, it is usable to express the applicability of expressions either in a actual parameters list or in an instruction.
Finally, we consider the variant of CF grammars allowing regular expressions in the right parts of the productions. Most traditional grammars of Pascal, starting from the first one in [6], use CF productions extended with regular expressions (called Extended BNF or EBNF), graphically presented as syntax diagrams. It is known from experience that EBNF descriptions are often smaller than BNF ones. The same approach could be applied to ALD rules too, as originally proposed in [1], with the difference that local testability sets [9] are used instead of general regular expressions. This direction has to be investigated in details.
In our opinion, the empirical and quantitative analysis reported here strengthens the qualification of the ALD model as an alternative technique for specifying languages. Of course other aspects in addition to descriptional complexity ought to be investigated, such as parsing and grammar inference, which could provide an interesting method for specifying a language by means of examples.

Acknowledgements

To V. Braitenberg and to F. Pulvenmüller for stimulating discussions on the role and potential of associative language models in brain theory. To A. Cherubini for important contributions to the study of the formal properties of the ALD family. To the referee for costructive criticism.

References

[1] S. CRESPI REGHIZZI , V. BRAITENBERG, "Towards a brain compatible theory of language based on local testability", in V. Braitenberg and F.J. Radermacher (eds), "Interdisciplinary approaches to a new understanding of cognition and consciuosness", to appear.

[2] A. CHERUBINI, S. CRESPI-REGHIZZI, P. SAN PIETRO, "Languages based on structural local testability", in C. S. Calude and M.J. Dinnen (eds.), *Combinatorics, Computation and Logic*, Proceedings of DMTCS99, Auckland, New Zealand, 18-21 January 1999, Springer-Verlag.

[3] S. CRESPI REGHIZZI, V. BRAITENBERG, "Towards a brain compatible theory of language based on local testability", in V. Braitenberg and F.J. Rödermacher (eds), *Interdisciplinary approaches to a new understanding of cognition and consciousness*, to appear.

[4] S. CRESPI REGHIZZI, G. GUIDA, D. MANDRIOLI, "Non-counting context-free languages", *Journ. ACM*, 25 (1978), 4, 571-580.

[5] A. EHRENFEUCHT, G. PAUN AND G. ROZENBERG, "Contextual grammars and formal languages", *Handbook of formal languages* (Eds. G.Rozenberg, A.Saloma), Vol.II, Ch.6, Springer (1997), 237-290.

[6] K. JENSEN AND N. WIRTH, *PASCAL - User manual and report*, LNCS, 18, 1974.

[7] S. MARCUS, "Contextual grammars and natural languages", *Handbook of formal languages* (Eds. G.Rozenberg, A.Saloma), Vol.II, Ch.6, Springer (1997), 215-132.

[8] R. MCNAUGHTON, *Parenthesis Grammars*, Journal of the ACM, Vol. 14, No. 3, 490-500, July 1967.

[9] R. MCNAUGHTON AND S. PAPERT, *Counter-Free Automata*, MIT Press, Cambridge, Mass., 1971.

[10] S. SIPPU, E. SOISALON-SOININEN, *Parsing Theory*, Springer-Verlag (1988).

Appendix: the syntax of the programming language Pascal

Notation Tokens are used for identifiers and constants: *id* stands for an identifier, while *cst* is a numerical constant. In a pattern p, the notation [w] denotes $w \cup \varepsilon$ (i.e. w is an optional part of the pattern p). This is not to be confounded with the square brackets used to separate the context from the pattern in an ALD production (e.g. in x[p]y).
The rules have the following structure:

L	P	R

Where P is the set of patterns of the rule; L is the set of left contexts and R is the set of right contexts. This denotes the set of rules: $\{<x,p,y> \mid <x,y> \in L \times R \wedge p \in P\}$. L or R = ∅ means "don't care". We sometimes use the usual set operations to denote some of the left/right context sets. Quoted strings denotes strings with special symbols (e.g. "[" means the character [). ~P stands for the complementary of set P: ~P := $\{w \in \Sigma^k \mid w \notin P\}$. The context sets are: CompOp = {=, >, <, >=, <=, <>}, AddOp = {+, -, or}, MultOp = {*, /, div, mod, and} and GenOp = CompOp ∪ AddOp ∪ MultOp.

⊥	program id [(id Δ)] ; Δ begin Δ end .	
".id", "]", ^	ε .id Δ ^ Δ "[" Δ "]" Δ	
with	[Δ ,] id [.id Δ] [Δ ,] id ^ D [Δ ,] id "[" Δ "]" Δ	
of	[Δ ;] Δ cst : Δ	end, cst
	ε cst , [Δ]	cst :
	[const Δ ;][type Δ ;][var Δ ;][procedure id Δ ;][function id Δ ;]	begin
const, "= cst ;"	id = cst [; Δ]	
type	[Δ ;] id = [^] Δ id [Δ ;] id = [^] Δ record Δ end	
var, record	[Δ ;] Δ : [^] Δ id [Δ ;] Δ : [^] Δ record Δ end	~{:}
:, of, =, ^	[packed] array "[" Δ "]" of Δ file of Δ set of Δ ε	id, record
array "["	[Δ ,] (id Δ) [Δ ,] id [Δ ,] cst .. cst	
set of	id (id Δ) cst .. cst	
(id	ε [Δ], id	
	id [, Δ]	:
procedure id	[(Δ)] ; Δ begin Δ end [; procedure id Δ] [(Δ)] ; Δ begin Δ end ; function id Δ	
function id	[(Δ)] : id ; Δ begin Δ end [; function id Δ] [(Δ)] : id ; Δ begin Δ end ; procedure id Δ	
"function id (", "procedure id ("	[Δ ;] [var] Δ : Δ id [Δ ;] [var] Δ : Δ record Δ end	
begin, ";", do, then, else, repeat, ":"	[cst :] id [. id Δ] := Δ [; Δ] [cst :] id ^ Δ := Δ [; Δ] [cst :] id "[" Δ "]" Δ := Δ [; Δ] [cst :] id (Δ) [; Δ] [cst :] for id := Δ to Δ do Δ [; Δ] [cst :] if Δ then Δ [else Δ] [; Δ] [cst :] while Δ do Δ [; Δ] [cst :] repeat Δ until Δ [; Δ] [cst :] goto cst [; Δ] [cst :] case Δ of Δ end [; Δ] [cst :] with Δ do Δ [; Δ]	";", end, else, until
~{function, procedure} id "(" ∪ {",", "id ["}	Δ, Δ	")","]"
↑ ∪ {:=, if, while, until, case} ∪ (GenOp ∪ {:=, ",", "("})."("}	Δ CompOp Δ	";", end, else, until, then, do, ")", "]", of}
↑ ∪ CompOp ∪ AddOp	{+,-} Δ [{+,-}] Δ AddOp Δ	↑ ∪ CompOp
↑ ∪ MultOp	Δ MultOp Δ	↑ ∪ AddOp
↑	[not] cst [not] id [. id Δ] [not] id ^ Δ [not] id "[" Δ "]" Δ [not] id (Δ) (Δ)	↑ ∪ MultOp