ASSOCIATIVE DEFINITION OF PROGRAMMING LANGUAGES[1]

Stefano Crespi Reghizzi[2], Matteo Pradella and Pierluigi San Pietro
Politecnico di Milano,
Dipartimento di Elettronica e Informazione,
P.za L. da Vinci 32, I-20133 Milano
e-mail: {crespi, pradella, sanpietr}@elet.polimi.it

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 is studied. A complete ALD grammar of Pascal testifies of the practicality of the ALD approach.

Keywords: Context Free Grammars, Syntax, Associative Grammars, Grammar Size, Context Inheritance, Descriptional Complexity, Local Testability

1. Introduction

In spite of their widespread use for defining computer languages, context-free (CF) grammars have shortcomings that prompted the search of alternative models. The recently proposed [2, 3] 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 the notion of local testability and context association. An ALD consists of purely terminal patterns accompanied by contextual constraints; it therefore qualifies as a "pure" grammar in the sense of [4], since it does not use nonterminals.

The ALD language family though smaller than the CF family, is here shown to be empirically adequate for defining useful computer languages. There are two main features of CF languages which exceed the capacity of ALD. The first one is counting: ALD languages are Non-Counting (NC) in the sense of [5], that is they may not

[1]Work supported by MURST-Cofinanziamento 9801204372/2 1999 and by CNR-CESTIA.
A preliminary version of this paper was presented at DCAGRS'99 [1].
[2]Also with Università della Svizzera Italiana, Ist. di Tecnologie della Comunicazione.

use numerical congruences for discriminating valid from invalid strings. The ability of CF grammars (and of course finite state automata) to perform modulo counting is clearly extraneous to the syntax of computer and natural languages. The second missing feature has to do with the phenomenon of synonymity, and is more difficult to describe. Two constructs which occur in two contexts which are identical up to an unbounded length cannot be discriminated using ALD. One could imagine that this lack of discriminatory capacity could jeopardize the new model: this paper shows to the contrary that useful computer languages can be defined by ALD. This work supplements in a practical sense the theoretical proof [3] that the Hardest CF language of Greibach [6] is an ALD language.

Incidentally, the ALD investigation was initially motivated by the search for a language model that would be consistent with modern brain theory [7], 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. Here we provide some evidence of ALD suitability for practical language definition, first by using ALD for describing a real programming language - Pascal [8], then by comparing the descriptional complexity of CF and ALD grammars.

Section 2 provides the basic definitions. Section 3 exhibits a large piece of empirical evidence, the ALD grammar of Pascal. In Section 4 simple bounds on the descriptional complexity of the two models are derived. Section 5 considers multi-level structures that occur in technical languages (mark-up languages or expressions with operator precedence), and compares their complexity using ALD and CF rules. It also observes the duality between the combinatorial growth of ALD permissible contexts and the growth of CF productions in copy-free grammars, and that context inheritance may reduce the complexity by the same amount as the introduction of the CF copy rules. In the conclusion, 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 labeled by Δ and whose leaves have labels in Σ is called a *(stencil) tree*. It is composed by juxtaposed subtrees having height one and leaves with labels in $\Sigma \cup \{\Delta, \epsilon\}$, called *constituents* (ϵ stands for the null string). The *frontier* of a tree T (of a constituent K_i) is denoted by $\tau(T)$ (resp. $\tau(K_i)$).

The next rather technical definition formalizes the notions of left and right contexts of a constituent.

Definition 1 (left/right contexts) *For an internal node i of a tree T, let K_i and T_i be resp. the constituent and the maximal subtree of T having root i. Introduce a new symbol $\omega \notin \Sigma \cup \{\perp, \Delta\}$, and consider the tree T_{K_i} obtained by replacing in T the subtree T_i with ω. Consider the frontier of T_{K_i}, $\tau(T_{K_i}) = s\omega t$ with $s, t \in \Sigma^*$. The*

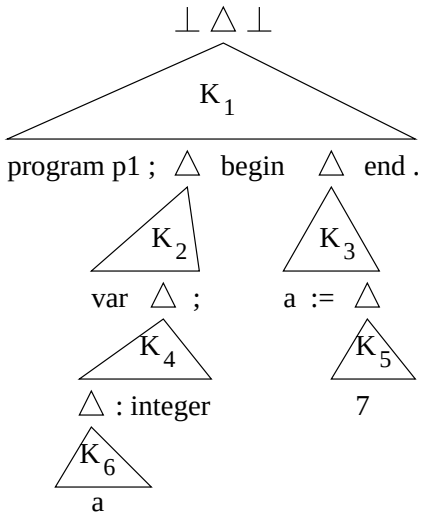

Figure 1: A stencil tree with six constituents schematized by triangles

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:
$\perp$ `program p1; var a : integer; begin`
and the right context of K_5 is:
`end.` $\perp$
Next comes the main definition.

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

Sometimes we use the notation (x, y) for denoting the pair of left and right contexts of a rule $x[p]y$.

An ALD provides a set of test conditions for checking the validity of a tree, namely that the contexts of each constituent are consistent with the permissible contexts of some rule. Therefore, an ALD is a device for defining a set of trees and a string language, corresponding to the frontiers of the trees.

Definition 3 (constituent matched by a rule, valid trees, tree and string language, equivalence and structural equivalence) *A rule* $u[z]v$ *matches a constituent* K_i *in* T *if* $z = \tau(K_i)$*,* 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*. A tree* T *is* valid for an ALD A *if each constituent of* T *is matched by a rule of* A*.*

The (stencil) tree language $T_L(A)$ *defined by an ALD A is the set of all 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.*

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 controlled by context permissibility.

For conciseness, we adopt the following notations to factorize common patterns and contexts. Two (or more) rules with the same pattern $x_1[z]y_1$ and $x_2[z]y_2$ can be merged into the complex rule: $x_1|x_2[z]y_1|y_2$, which can be merged with a rule having the same contexts $x_1|x_2[w]y_1|y_2$ into the complex rule $x_1|x_2[z|w]y_1|y_2$.

Moreover, if a permissible context is irrelevant, it can be omitted and replaced by ϵ (meaning "don't care"). Finally, in a pattern p, the notation $\langle w \rangle$ stands for $w \cup \epsilon$ (i.e. w is an optional part of the pattern p).

Examples The language $\{a^n c b^n \mid n \geq 1\}$ is defined by the ALD rules: $\bot[a\Delta b]\bot$; $a[a\Delta b]b$; $a[c]b$.

Both contexts can be dropped from the first two rules, and either the right or the left context can be omitted from the last rule, obtaining the equivalent description: $[a\Delta b]$; $a[c]$.

Some rules may be *useless*: e.g. adding the rule $[aa]a$ to the previous ALD, does not change the defined language. That is because its right context does not match any constituent.

Obviously, all 1-nonterminal context-free languages are ALD. For instance, the Dyck language over the alphabet $\Sigma = \{a, a'\}$ is defined by the ALD rules: $[a\Delta a'\Delta]$; $[a\Delta a']$; $[aa'\Delta]$; $[aa']$; $[\epsilon]$ where all contexts are "don't care", or more concisely by: $[a\langle\Delta\rangle a'\langle\Delta\rangle]$; $[\epsilon]$.

Ambiguity can occur in ALDs much as in CF grammars. For example, the following rules ambiguously define the Dyck language: $[\Delta\Delta]$; $[a\Delta a']$; $[\epsilon]$.

For a rule $x[z]y$ we introduce some integer attributes:

- the *left* (resp. *right*) *degree* of a rule is $|x|$ (resp. $|y|$);
- the *degree* of a rule is $max\{|x|, |y|\}$;
- the *degree* of an ALD is the maximum of the degrees of its rules.

A tree language T is of degree k, if there exists an ALD A of degree k such that $T = T_L(A)$; similarly for a string language. The degree induces an infinite hierarchy of ALD language families [3].

It is sometimes convenient to assume that contexts have the same length and that all rules are useful, two properties next defined.

Definition 4 (homogeneous, reduced ALDs) *An ALD A of degree k is* homogeneous *if for every rule $x[p]y \in A$,*

- $x \in \{\Sigma^k \cup \bot\Sigma^j \mid 0 \leq j < k\}$ *and*
- $y \in \{\Sigma^k \cup \Sigma^j\bot \mid 0 \leq j < k\}$.

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

We mention from [3] some relevant properties of ALD and relations between ALD and CF models.

Statement 1 *For any ALD there exists a structurally equivalent CF grammar (i.e., which generates the same set of stencil trees and hence also the same string language).*

Let k be the degree of the ALD (which can be assumed homogeneous). The proof is based on the idea of assigning to every ALD subtree T a syntax category (i.e. a nonterminal) characterized by four parameters: (*le, fi, la, ri*), where *le* and *ri* are resp. the left and the right context of length k of the subtree; *fi* and *la* are the first and the last k terminal characters of $\tau(T)$ (see Definition 1).

Other relevant facts are:

- The ALD family of languages is strictly included in the CF family.
  For instance the language $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$ is not ALD. This fact is proved using a *swapping lemma*, which states that subtrees which may occur in the same unbounded contexts may be interchanged.
- Yet the ALD family contains the *hardest CF language of Greibach* [6].
- ALD tree languages are Non-Counting in the sense of [5].
- The Locally Testable regular languages (in the strict sense of [9]) are included in the ALD family. (It is an open problem whether all regular languages are ALD.)

3. Applying Associative Language Descriptions: the case study 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 that will be briefly considered in Section 5.

It was somewhat 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 (of course, semantic constraints exceed the descriptive capacity of both models).

Following the classical separation of lexicon and syntax, we assume that the source text has been already tokenized by a lexical analyzer or scanner. This allows us to focus on the syntactic aspects of the language. The tokens of Pascal are identifiers,

keywords, numerical constants, comments, etc. It could be of some interest to observe that the lexicon, which is usually defined using regular expressions or finite automata, can be defined by an ALD, even if it is unknown whether all regular languages are ALD. For brevity sake we do not dwell on the lexical definitions and jump straight to the syntax.

Shorthands: rule blocks and context inheritance

The ALD of Pascal and of other complex languages, though straightforward to derive, would be rather cumbersome to write if only "simple" ALD rules were to be used. In order to shorten the ALD and to highlight constructs which are equivalent with respect to context, we use two convenient devices: *Cartesian products* and *context inheritance*.

Cartesian products are an effective method for simplifying complex ALD rule definitions. For instance, a typical Pascal instruction can occur, among the others, inside the contexts: $(begin, end)$, $(then, else)$, $(repeat, until)$, $(;,;)$ and many combinations of them, like $(repeat, ;)$. A defining ALD should consider *every* possible combination in its definition, resulting often in contexts that are long, complex and hard to read.

The Cartesian product notation permits more readable ALD definitions, and is based on the simple idea of implicitly considering every combination of the left and right contexts. This sometimes leads to inclusion of unwanted combinations (e.g. $(repeat, else)$), but in many practical cases this is not a problem, since they are useless and therefore cannot be exploited. In Section 4 we introduce an effective theoretical condition to use Cartesian products.

Definition 5 (rule block) *A* rule block *is a triple denoted by* $\boxed{L \parallel P \parallel R}$ *where* P *is a set of patterns,* L *is a set of left contexts and* R *is a set of right contexts. The rule block denotes the set of ALD rules:* $\{x[p]y \mid (x, p, y) \in L \times P \times R\}$*. A void* L *or* R *stands for* $\{\epsilon\}$*, or "don't care". A set of rule blocks denotes the union of the ALDs denoted by each rule block.*

We will use some more shorthand notations, necessary to improve readability.

The | separator is used in context set even to denote union. For instance, let A be a set and a, b be strings: the context $A|a|b$ denotes the set $A \cup \{a, b\}$.

An example of a rule block is the following:

of	$\langle\Delta\rangle$ cst : Δ	end | cst

It denotes the two ALD rules: *of* $[\langle\Delta\rangle$ *cst :* $\Delta]$*end* and *of* $[\langle\Delta\rangle$ *cst :* $\Delta]$*cst*.

Context inheritance has the same purpose and effect as renaming productions in CF grammars. Productions such as $expression \rightarrow term$ in the grammar
$expression \rightarrow factor * term + expression$
$expression \rightarrow factor * term$
$term \rightarrow factor * term$
$term \rightarrow factor$
etc.

reduce the size of the grammar by stating that wherever an *expression* occurs a *term* may occur as well (subcategorization).

In the ALD of Pascal (see the appendix) context inheritance conveniently factorizes repeated context sets of arithmetic or Boolean expressions.

Definition 6 (ALD with inheritance) *Extend rule blocks by allowing a new symbol* $\Uparrow$ *in the contexts, i.e., a rule block* $\boxed{L \| P \| R}$ *is such that* $L \in \wp(\Uparrow \cup (\bot\Sigma^*) \cup \Sigma^*)$, *and* $R \in \wp(\Uparrow \cup (\Sigma^*\bot) \cup \Sigma^*)$. *An* ALD with inheritance S *is a sequence of* $h \geq 1$ *rule blocks, such that the* $\Uparrow$ *does not occur in the contexts of the first rule block. For the* i*-th rule block* $\boxed{L_i \| P_i \| R_i}$ *of* S, $1 \leq i \leq h$, *let* $L'_i = (L_i \cup L'_{i-1} - \{\Uparrow\})$ *if* $\Uparrow \in L_i$, $L'_i = L_i$ *otherwise; let* $R'_i = (R_i \cup R'_{i-1} - \{\Uparrow\})$ *if* $\Uparrow \in R_i$, $R'_i = R_i$ *otherwise. Then, the ALD denoted by* S *is the set of rule blocks* $\left\{ \boxed{L'_i \| P_i \| R'_i} \mid 1 \leq i \leq h \right\}$.

The ALD definition of Pascal strongly uses inheritance. For instance the definition of expressions is the following:

:= | if | ...	Δ CompOp Δ	; | end | else | ...

⇑ | CompOp | AddOp	{+ | -} Δ | ⟨{+ | -}⟩ Δ AddOp Δ	⇑ | CompOp

⇑ | MultOp	Δ MultOp Δ	⇑ | AddOp

⇑	⟨ not ⟩ cst | ⟨ not ⟩ id (Δ) | (Δ) | ...	⇑ | MultOp

This is taken from the appendix, rules from 20 to 23, and omits for simplicity many of the contexts of the first rule, necessary for using expressions within instructions.

Length of the contexts for Pascal

Concerning the length of the contexts which are needed for Pascal, we found that for most rules, the degree one[3] suffices, with a majority of rules having don't cares in at least one context. Quite frequently the left context is longer than the right one; a fact which agrees with our intuition that since strings are parsed from left to right, past tokens are more important than look-ahead tokens for determining grammaticality.

For example, the following two rule blocks define procedures and functions (Appendix, rules 13 and 14):

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	

[3]Of course here we count tokens not single characters of the keyboard.

The maximum degree 3 is required in one localized construct, the specification of parameters of functions or procedures. The permissible left contexts are the two strings `function id (` and `procedure id (` – the right context is "don't care". The rather long left context is needed to distinguish the construct (formal parameters declaration – Appendix, rule 15):

function id (|	⟨Δ ;⟩ ⟨var⟩ Δ : Δ id |	
procedure id (	⟨Δ ;⟩ ⟨var⟩ Δ : Δ record Δ end	

from the construct (actual parameters list – Appendix, rule 19):

¬{function | procedure} id (| , | id [	Δ , Δ	) |]

The last context description expression is based on negative examples. In fact `¬{function | procedure} id (` means any string `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 contexts, instead of the permissible ones. This is analogous to the description of exceptions, instead of regularities.

Don't cares occur frequently, e.g., 3 being the degree of this ALD, the context $(with, \epsilon)$, used in the rule block (Appendix, rule 18):

with	⟨Δ ,⟩ id ⟨.id ⟨Δ⟩⟩ | ⟨Δ ,⟩ id ^ ⟨Δ⟩ | ⟨Δ ,⟩ id [Δ] ⟨Δ⟩	

stands for the set of contexts $\{(u.with, v) \mid u \in \Sigma^2 \cup \perp\Sigma, v \in \Sigma^3 \cup \Sigma^2\perp\}$.

By comparing the ALD with the CF grammar of Pascal, that is reproduced in many textbooks, the reader will find the sizes to be about the same.

A disclaimer: 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.

Designing a complex ALD

In this section we collect some notes deriving from the experience of designing a complex ALD, such as the one for Pascal.

First, we proceeded in a top-down fashion, constructing the syntactic trees of Pascal programs, from simple to more complex ones, disregarding of course the names of nonterminal symbols, which do not exist for an ALD.

Usually, it is natural to start with short, often asymmetric contexts. In the case of Pascal, we started with a left context of length one, and zero length right contexts. After inserting a rule block, it is trivial to compute context intersections with the already defined ones. (An automatic tool for checking this could be designed).

Consider an example. Every Pascal program has the following structure (as stated in the previous section, the left $\perp$ context denotes the top rule) – Appendix, rule 1:

⊥	program id ⟨(id Δ)⟩ ; Δ begin Δ end .	

The first place holder is used for the program parameters, if any. The second delta stands for every kind of definition (variables, functions, etc.). The third delta denotes the main body of the program. Next we write the rules for the place holders above.

A straightforward candidate definition for the parameters list of the first delta is:

id	⟨Δ⟩ , id	

While a quite natural description of the main program definitions is:

;	⟨const Δ ;⟩⟨type Δ ;⟩⟨var Δ ;⟩⟨procedure id Δ ;⟩⟨ function id Δ ;⟩	

So far all the used contexts are different, therefore no overlapping of contexts may cause over generalization. That is, we cannot use one rule instead of another, while building a stencil tree.

Let us continue with the rule defining a procedure. Now a problem arises, if we try to define procedures by a left context of length 1 and a don't care for right context:

id	⟨(Δ)⟩ ; Δ begin Δ end ⟨; procedure id Δ⟩	

The left context is *id*, which is the same of the program parameters list definition. The two constructs are thus interchangeable, causing an error because it should not be possible to put a procedure definition in a parameters list.

A simple solution is to enlarge the left or the right contexts by one. We choose to increase the left context, because enlarging the right context would not do: a permissible right context of the procedure definition is the semicolon, a widely used separator which is likely to cause confusion with other constructs.

Extending the two conflicting contexts we obtain the rules (Appendix, rules 2 and 13):

(id	⟨Δ⟩ , id	

procedure id	⟨(Δ)⟩ ; Δ begin Δ end ⟨; procedure id Δ⟩	

A similar problem arises in the definition of a list of assignment instructions:

begin | ;	id := Δ ⟨; Δ⟩	

The left semicolon conflicts with the main program definitions. To discriminate we use the right contexts (Appendix, rules 3 and 16):

;	⟨const Δ ;⟩⟨type Δ ;⟩⟨var Δ ;⟩⟨procedure id Δ ;⟩⟨ function id Δ ;⟩	begin

begin | ;	id := Δ ⟨; Δ⟩	end | ;

Overlapping contexts are not necessarily wrong, a case to be illustrated by the example of relational expressions.

A simplified definition of a relational expression is (from Appendix, rule 20):

if | := | . . .	Δ CompOp Δ	end | ; | else | . . .

where $CompOp = \{=, >, <, >=, <=, <>\}$ is a relational operator.

On the other hand an arithmetic expression is defined by (Appendix, rule 21):

AddOp | if | := | . . .	Δ AddOp Δ	CompOp | end | ; | else | . . .

where $AddOp = \{+, -, or\}$.

Both the left and the right contexts of relational and aritmetic expressions overlap, but this is correct. For instance, in an assignment both a comparison and a sum can occur (e.g. in $x := \Delta$, where Δ can be $y > 4$, or $y + 4$); similarly in an *if* statement both a relational expression and a Boolean one can occur, as in *if* $y > 4$, or *if a or b*.

This situation is ideal for context inheritance: The contexts of the arithmetic expression are a superset of the contexts of the relational expression, because a sum precedes a comparison in the operator hierarchy (e.g. in $y + 3 > z + 4$). In choosing

the context of a new rule block, the designer can take advantage of inheritance. As he/she writes down the tree of a typical Pascal program, inheritance can be used to specify the possibility of interchange between subtrees. It is quite convenient to re-use some complex contexts already described, by referencing them with $\Uparrow$.

Summarizing, here is a short description of the conceptual design steps:

1. Choose the current length for the contexts (typically one for the left context and zero for the right context).
2. For a certain language construct, write a rule block definition, taking the contexts of the current length. The contexts are computed using the previous rule blocks, according to the desired position of the place holder. Use inheritance, if the contexts of the previous rule block can be reused.
3. When a new rule (block) is introduced, check that context intersections w.r.t. every existing rule are empty.
4. If all intersections are void, then the current rule block is consistent with the ALD.
5. If the intersection with some rule block is not void, two cases arise:
   (a) The two constructs may legally occur in the same program context: the current rule block and the other one are interchangeable;
   (b) The two construct may never occur in the same program context. This is a symptom that the current context length is insufficient, and must be increased. This can be done by replacing a "don't care" with an actual context, or by effectively increasing the context length by one (left or right).

The design steps are iterated until all language structures have been successfully considered.

If the context length increases unboundedly, it is likely that the tree language to be designed is not an ALD language (but the same language could be ALD for a different tree structure).

It is important to understand why the previous algorithm must be applied iteratively. The creation of a new rule block can offer in its pattern applicability for some previous rule. Therefore, the contexts sets of the previous rule have to be recomputed, considering this new rule.

It should be noted that long contexts can be simplified, if they are redundant. For instance, we built a rule block with left and right contexts of length 2. Then we observed, by computing the context set intersection, that the right context is not necessary (as in many rules reported in the Appendix); therefore, we discarded it. This simplification could be introduced after step 4, or after step 5.

4. 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 both in linguistics and in compiler technology the structure to be assigned to a sentence is usually imposed by semantics.

Firstly, we state some definitions and properties, that are quite useful in the comparison with CF grammars. To formalize the structural properties of CF grammars we take the classical approach of Mc Naughton [10] based on parenthesis grammars.

Let $G = (V, \Sigma, P, S)$ be a CF grammar, where V are the nonterminals, P the productions and S the axiom. We assume the parentheses '(' and ')' and the terminator $\perp$ not to be in Σ. To be consistent with the notation introduced for ALD, we also assume that G derives words of the form $\perp\ z\ \perp$, where $z \in \Sigma^*$.

Definition 7 (parenthesis grammars and languages) *The* CF parenthesis grammar *associated with G is $G' = (V, \Sigma \cup \{(,)\}, P', S)$, where $P' = \{B \to (\alpha) \mid B \to \alpha \in P \wedge \alpha \notin V\} \cup \{B \to C \mid B \to C \in P \wedge C \in V\}$ (from [10]). Likewise, the parenthesis ALD grammar A' associated with an ALD 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 defined by G' or A', respectively.*

Two language definitions 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 copy rules, because they are a mere expedient for reducing grammar size.

Statement 2 *For every ALD there exists a structurally equivalent, homogeneous reduced (Def. 4) ALD.*

Proof. Let k be the degree of a given ALD A. A homogeneous, structurally equivalent ALD is $A'' = \{ux[z]yv \mid x[z]y \in A \wedge ux \in \Sigma^k \cup \{\perp\ \Sigma^j \mid 0 \leq j < k\} \wedge yv \in \Sigma^k \cup \{\Sigma^j\ \perp \mid 0 \leq j < k\}\}$, which may contain unusable rules which can be eliminated without changing the language, giving a reduced ALD. □

Next we show that for a given tree language the ALD is unique, provided that the degree is fixed and the rules are homogeneous. This uniqueness property does not hold for CF grammars: two CF grammars generating the same tree language, i.e. structurally equivalent, may be different.

Statement 3 *If A' and A'' are two structurally equivalent reduced homogeneous ALDs of the same degree, then $A' = A''$.*

Proof. Suppose by contradiction that in $A' - A''$ there is a rule $u[p]v$, which (being A' reduced) is used in a tree $T \in L_T(A')$. Since $L_T(A') = L_T(A''), T \in L_T(A'')$, which is absurd since $u[p]v \notin A''$ and A' and A'' are homogeneous of the same degree. □

Let us consider Statement 1. Observing that the nonterminals of the equivalent CF grammar are 4-tuples (*le, fi, la, ri*), the next statement follows immediately.

Corollary 1 *For any ALD of degree k, there exists a structurally equivalent CF grammar having at most* $|\Sigma|^{4k}$ *nonterminals.*

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 need the following definition from [10].

Definition 8 (McN-patterns, equivalent and useless nonterminals, reduced CF grammars) *For a grammar G, 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[4]. *Moreover, for* $B \in V$ *denote with* $\beta[B]$ *the result of the replacement of the placeholder in* β *with B.*

Two nonterminals $B, C \in V$ *are* equivalent *iff* $\forall \beta (S \rightarrow^+ \beta[B] \iff S \rightarrow^+ \beta[C])$. *A nonterminal B is* useless *iff there exists no* β *such that* $S \rightarrow^+ \beta[B]$ *or there exists no* $t \in \Sigma^*$ *such that* $B \rightarrow^+ t$.

A CF grammar is reduced *if it does not contain equivalent or useless nonterminals.*

It is well-known that for any CF grammar there exists an equivalent reduced grammar.

Next, we consider the unbounded and bounded contexts of a nonterminal and a mapping from the right-hand sides of CF productions to the patterns of the ALD rules.

Definition 9 (context and k-context of a nonterminal) *The context of a nonterminal B of a CF grammar G is* $Con(B) = \{(u, v) \mid u \in \perp \Sigma^*, v \in \Sigma^* \perp \wedge S \rightarrow^+ uBv\}$.

The context of degree $k \geq 1$ *(or* k-context*) is* $Con_k(B) = \{(u', v') \mid u' \in \Sigma^k \cup \perp\Sigma^j, v' \in \Sigma^k \cup \Sigma^j\perp$, *with* $0 \leq j < k$; $(u, v) \in Con(B)$, u' *is a suffix of u,* v' *is a prefix of* $v\}$.

Definition 10 (pattern set) *Let* $h : (V \cup \Sigma)^* \rightarrow (\Delta \cup \Sigma)^*$ *be the alphabetic homomorphism defined as* $h(B) = \Delta$ *for* $B \in V$, $h(v) = v$ *for* $v \in \Sigma$. *For a nonterminal* $B \in V$, *the* pattern set of B *is defined as* $Pat(B) = \{h(\alpha) \mid B \rightarrow \alpha \in P \wedge \alpha \notin V\}$.

The last 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.

Statement 4 (relations between a CF grammar and its structurally equivalent ALD)

(i) $\forall B \rightarrow \alpha \in P, \alpha \notin V$, *there exists in A a rule* $u[h(\alpha)]v$.

(ii) $(u, v) \in Con_k(B)$.

[4]It corresponds to the definition of pattern in [10], which has nothing to do with the "pattern" of ALD.

(iii) For every pair of rules $B \to \beta \in P, C \to \gamma \in P, \beta, \gamma \notin V$, let $u[h(\beta)]v, x[h(\gamma)]y$ be in A. If there exists in P a copy rule $B \to C$ then 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 $D \to \alpha \in P, \alpha \notin V$, is required as pattern of A. Therefore, the set of patterns of A equals $\bigcup_{D \in V} Pat(D)$. (ii) Since G is reduced, $h(\beta)$ and $h(\gamma)$ are required as patterns of A. (iii) 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 $A = \{(Con_k(B), Pat(B)) \mid B \in V\}$.

A first comparison of the relative size of ALD and CF grammars follows immediately from the definition of the homomorphism h and of the set $Pat(B)$, $B \in V$:

Statement 5 *For every nonterminal $B \in V$, $|Pat(B)| \leq |\{\alpha \mid B \to \alpha \in P\}|$.*

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.

Statement 6 *For an ALD $A = \{(Con_k(B), Pat(B)) \mid B \in V\}$, the cardinality of the set $\{Con_k(B) \mid B \in V\}$* (distinct permissible context sets) *is $|V|$.*

Proof. Let $B, C \in V$. It is known ([10]) 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 follows 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 reduced, there are no equivalent nonterminals. Therefore, we have: $\forall B, C \in V (B \neq C \Rightarrow Con_k(B) \neq Con_k(C))$, which is our statement. □

Cartesian Products

Cartesian products of left and right context sets are usually extensively exploited (see Section 3). This simple technique is enabled by the next statement.

Consider a CF grammar as in Statement 1, and its structurally equivalent ALD of degree k. Let $\mathcal{L}(Con_k(B))$ (resp. $\mathcal{R}(Con_k(B))$) denote the projection of $Con_k(B)$ on the left (resp. right) component.

Statement 7 *Consider a nonterminal B and the corresponding ALD rule $(Con_k(B), Pat(B))$. If for every nonterminal $C \neq B$ the following condition holds $Con_k(B) \cap Con_k(C) = (\mathcal{L}(Con_k(B)) \times \mathcal{R}(Con_k(B))) \cap Con_k(C)$, then the rule $(Con_k(B), Pat(B))$ can be replaced by $(\mathcal{L}(Con_k(B)), Pat(B), \mathcal{R}(Con_k(B)))$, 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. An example can be found in the definition of the array, file and set constructs of Pascal.

5. Size 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 dimension of a grammar strongly depends 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. An analogous technique for ALD is based on the concept of context-inheritance (introduced in Section 3).

The next definitions (adapted to ALD from [11]) are needed to compare sizes.

Definition 11 (size, norm) *The* size $|H|$ *of a CF/ALD grammar H is the sum of the lengths of their rules, considering only the symbols in $Sym = V \cup \Sigma \cup \{\epsilon\}$ for the CF case and in $Sym = \{\Delta, \bot, \Uparrow, \epsilon\} \cup \Sigma$ 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 latter measure in our case is no more informative than the former, because in our examples $|V|$ is always proportional to $|\Sigma|$, 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 we will only compare the *size*.

The abstract syntax structures to be compared occur in various incarnations such as:

- Parentheses structures featuring different pairs of parenthesis, ranked into a hierarchy. The typical example is HTML, an instance of the generalised mark-up languages SGML or XML.
- 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.

Actually, the simplified version of the first family that we consider here is regular. Nonetheless, its inherent hierarchical structure is more naturally and concisely described using tree structures or CF productions, rather than by Chomsky's type 3 productions.

To focus on one concrete case, consider the language with $n > 1$ pairs of parentheses ranked by level, in which a pair of level i may only contain parentheses of strictly lower levels. The terminal alphabet is $\Sigma = \{<_1, >_1, \ldots, <_n, >_n\}$. An instance of a valid string, with $n = 3$, is: $<_1>_1<_3<_2>_2<_1>_1>_3<_2>_2$.

The ALD of degree 1 has n rules and size $1/2n^2 + 9/2n$:

$$[\epsilon \mid <_n \langle\Delta\rangle >_n \langle\Delta\rangle]\bot;$$

$$[<_{n-i} \langle\Delta\rangle >_{n-i} \langle\Delta\rangle]\bot \mid >_n \mid \ldots \mid >_{n-i+1} \quad (where\ 1 \leq i \leq n-2); \tag{1}$$

$$[<_1>_1 \langle\Delta\rangle]\bot| >_2 | \ldots | >_n$$

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 $\Uparrow$ (see Section 3).

Using inheritance the previous ALD becomes:

$$\begin{aligned} &[\epsilon \mid <_n \langle\Delta\rangle >_n \langle\Delta\rangle]\bot \\ &[<_{n-i} \langle\Delta\rangle >_{n-i} \langle\Delta\rangle] \Uparrow | >_{n-i+1} \quad (where\ 1 \leq i \leq n-2) \\ &[<_1>_1 \langle\Delta\rangle] \Uparrow | >_2 \end{aligned} \tag{2}$$

Note that the size of this grammar is linear with respect to n: the size of (2) is $6n - 1$.

We now compare (2) with different CF grammar versions for the same language.

First, we consider a structurally equivalent CF grammar, *reduced* and *copy-free* with N_n as axiom:

$$\begin{aligned} &N_i \rightarrow <_i N_{i-1} >_i N_n \mid <_{i-1} N_{i-2} >_{i-1} N_n \mid \ldots \\ &\mid <_1>_1 N_n \mid \epsilon \ (where\ 1 \leq i \leq n) \end{aligned} \tag{3}$$

The size of this grammar is $2n^2 + 3n$. This size is comparable with the size of the ALD (1).

We can build a smaller CF grammar, using copy rules:

$$\begin{aligned} &N_1 \rightarrow <_1>_1 N_n \mid \epsilon \\ &N_i \rightarrow <_i N_{i-1} >_i N_n \mid N_{i-1} \ (where\ 2 \leq i \leq n) \end{aligned} \tag{4}$$

The size is $6n - 1$, exactly the same of (2), the ALD with inheritance.

Analogous quantitative results hold for examples of the ubiquitous case of expressions with operators ranked by precedence. In this abstract syntax paradigm we admit one type of parenthesis.

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

$$\begin{aligned} &(\mid \bot[\Delta \#_n \Delta] \\ &\Uparrow \mid \#_n[\Delta \#_{n-1} \Delta] \quad \ldots \\ &\Uparrow \mid \#_2[\Delta \#_1 a \mid a \mid (\Delta)] \end{aligned} \tag{5}$$

Clearly, its abstract structure is very similar to that of (2). Correspondingly, a structurally equivalent CF grammar with copy rules is the following:

$$\begin{aligned} &N_1 \rightarrow N_1 \#_n a \mid a \mid (N_n) \\ &N_i \rightarrow N_i \#_i N_{i-1} \mid N_{i-1} \ (where\ 2 \leq i \leq n) \end{aligned} \tag{6}$$

and its structure is almost the same of (4).

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

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

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

6. Conclusions

We have shown that a complete programming language can be defined using ALD expressions with rather short contexts. The relative sizes of ALD and CF definitions have been found to be essentially the same, for Pascal and for the hierarchical structures occurring in HTML and in other common constructs of computer languages. Another valuable property of the model is the essential uniqueness of the ALD defining a set of trees.

After these findings, it would be interesting to experiment with the use of ALD in compilers, and related applications, but for that we need efficient parsing algorithms. In principle the classical CF parsing methods could be used for ALDs' too, by first converting an ALD into a structurally equivalent CF grammar, then constructing a parser using a parser generator. But direct construction of deterministic parsers from an ALD definition is more attractive, since the mechanically-generated, equivalent CF grammar may be very large. Research in this direction is in progress.

Finally we mention that the ALD design steps outlined at the end of Section 3 can be viewed as the blueprint of a simple grammar inference procedure. Grammar inference is the process that constructs a grammar by examining the examples of sentences (and non-sentences) provided by an informant. There are real situations, as in marked up XML documents, where a language grammar is not available, but it has to be discovered by examining examples of structured sentences (trees).

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.

References

[1] S. Crespi Reghizzi, M. Pradella, P. San Pietro, *Conciseness of Associative Language Descriptions*, Proc. of International Workshop on Descritional Complexity of Automata, Grammars and Related Structures (DCAGRS'99), J. Dassow and D. Wotschke (eds.), July 20-23, 1999, p 99-108.

[2] A. Cherubini, S. Crespi Reghizzi, P. San Pietro, *Associative Language Descriptions*, to appear in *Theoretical Computer Science*.

[3] 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.

[4] 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.

[5] S. Crespi Reghizzi, G. Guida, D. Mandrioli, *Non-counting context-free languages*, Journ. ACM, 25 (1978), 4, 571-580.

[6] S. A. Greibach, *The Hardest Context-free Language*, SIAM J. Comp., 2, (1973), 304-310.

[7] S. Crespi Reghizzi, V. Braitenberg, *Towards a brain compatible theory of language based on local testability*, in C. Martin-Vide and V. Mitrana (eds) Grammars and Automata for String Processing: from Mathematics and Computer Science to Biology, and Back. Gordon and Breach, London, 2001.

[8] K. Jensen and N. Wirth, *PASCAL - User manual and report*, LNCS, 18, 1974.

[9] R. McNaughton and S. Papert, *Counter-Free Automata*, MIT Press, Cambridge, Mass., 1971.

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

[11] S. Sippu, E. Soisalon-Soininen, *Parsing Theory*, Springer-Verlag (1988).

Appendix: the syntax of Pascal

Remark: The degree of the ALD of syntax is measured assuming a token is a length one character. For instance the key word `begin` counts as a single character.
Tokens are used for identifiers and constants: *id* stands for an identifier, *cst* is a numerical constant. Names starting with a capital letter always denote sets.
The used context sets are:
$CompOp = \{=, >, <, >=, <=, <>\}$;
$AddOp = \{+, -, or\}$;
$MultOp = \{*, /, div, mod, and\}$;
$GenOp = CompOp \cup AddOp \cup MultOp$.

1. | $\perp$ | program id $\langle$(id Δ)$\rangle$; Δ begin Δ end . | |
2. | (id | $\langle\Delta\rangle$, id | |
3. | ; | $\langle$const Δ ;$\rangle\langle$type Δ ;$\rangle\langle$var Δ ;$\rangle\langle$procedure id Δ ;$\rangle\langle$ function id Δ ;$\rangle$ | begin |
4. | const \| = cst ; | id = cst $\langle$; $\Delta\rangle$ | |
5. | type | $\langle\Delta$;$\rangle$ id = $\langle\hat{}\rangle$ Δ id \| $\langle\Delta$;$\rangle$ id = $\langle\hat{}\rangle$ Δ record Δ end | |
6. | var \| record | $\langle\Delta$;$\rangle$ Δ : $\langle\hat{}\rangle$ $\langle\Delta\rangle$ id \| $\langle\Delta$;$\rangle$ Δ : $\langle\hat{}\rangle$ $\langle\Delta\rangle$ record Δ end | $\neg${:} |
7. | : \| of \| = \| ^ | $\langle$ packed $\rangle$ array [Δ] of Δ \| file of Δ \| set of Δ | id \| record |
8. | array [| $\langle\Delta$,$\rangle$ (id Δ) \| $\langle\Delta$,$\rangle$ id \| $\langle\Delta$,$\rangle$ cst .. cst | |
9. | of | $\langle\Delta$;$\rangle$ $\langle\Delta\rangle$ cst : Δ | end \| cst |
10. | | cst , $\langle\Delta\rangle$ | cst : |
11. | set of | id \| (id $\langle\Delta\rangle$) \| cst .. cst | |
12. | | id $\langle$, $\Delta\rangle$ | : |
13. | procedure id | $\langle(\Delta)\rangle$; Δ begin Δ end $\langle$; procedure id $\Delta\rangle$ \| $\langle(\Delta)\rangle$; Δ begin Δ end ; function id Δ | |

14.

function id	⟨(Δ)⟩ : id ; Δ begin Δ end ⟨; function id Δ⟩ | ⟨(Δ)⟩ : id ; Δ begin Δ end ; procedure id Δ	

15.

function id (| procedure id (	⟨Δ ;⟩ ⟨var⟩ Δ : Δ id | ⟨Δ ;⟩ ⟨var⟩ Δ : Δ record Δ end	

16.

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

17.

.id |] | ^	.id ⟨Δ⟩ | ^ ⟨Δ⟩ | [Δ] ⟨Δ⟩	

18.

with	⟨Δ ,⟩ id ⟨.id ⟨Δ⟩⟩ | ⟨Δ ,⟩ id ^ ⟨Δ⟩ | ⟨Δ ,⟩ id [Δ] ⟨Δ⟩	

19.

¬{function | procedure} id (| , | id [	Δ , Δ	) |]

20.

⇑ | := | if | while | until | case | {GenOp | := | , | (} (	Δ CompOp Δ	; | end | else | until | then | do |) |] | of

21.

⇑ | CompOp | AddOp	{+ | -} Δ | ⟨{+ | -}⟩ Δ AddOp Δ	⇑ | CompOp

22.

⇑ | MultOp	Δ MultOp Δ	⇑ | AddOp

23.

⇑	⟨ not ⟩ cst | ⟨ not ⟩ id ⟨ .id ⟨Δ⟩⟩ | ⟨ not ⟩ id ^ ⟨Δ⟩ | ⟨ not ⟩ id [Δ] ⟨Δ⟩ | ⟨ not ⟩ id (Δ) | (Δ)	⇑ | MultOp